Logical Methods in Computer Science
Vol. 5 (4:2) 2009, pp. 1–48
www.lmcs-online.org

Submitted Mar. 3, 2009
Published Dec. 17, 2009

A STEP-INDEXED SEMANTICS OF IMPERATIVE OBJECTS *

CĂTĂLIN HRIŢCU [a] AND JAN SCHWINGHAMMER [b]

[a] Department of Computer Science, Saarland University, Saarbrücken, Germany
e-mail address: hritcu@cs.uni-sb.de

[b] Programming Systems Lab, Saarland University, Saarbrücken, Germany
e-mail address: jan@ps.uni-sb.de

Abstract. Step-indexed semantic interpretations of types were proposed as an alternative to purely syntactic proofs of type safety using subject reduction. The types are interpreted as sets of values indexed by the number of computation steps for which these values are guaranteed to behave like proper elements of the type. Building on work by Ahmed, Appel and others, we introduce a step-indexed semantics for the imperative object calculus of Abadi and Cardelli. Providing a semantic account of this calculus using more 'traditional', domain-theoretic approaches has proved challenging due to the combination of dynamically allocated objects, higher-order store, and an expressive type system. Here we show that, using step-indexing, one can interpret a rich type discipline with object types, subtyping, recursive and bounded quantified types in the presence of state.

1. Introduction

The *imperative object calculus* of Abadi and Cardelli is a very small, yet very expressive object-oriented language [2]. Despite the extreme simplicity of its syntax, the calculus models many important concepts of object-oriented programming, as well as the often subtle interaction between them. In particular it raises interesting and non-trivial questions with respect to typing.

In contrast to the more common class-based object-oriented languages, in the imperative object calculus every object comes equipped with its own set of methods that can be updated at run-time. As a consequence, the methods need to reside in the store, *i.e.*, the store is *higher-order*. Moreover, objects are *allocated dynamically* and aliasing is possible. Dynamically-allocated, higher-order store is present in different forms in many practical programming languages (*e.g.*, pointers to functions in C and general references in SML), but it considerably complicates the construction of adequate semantic models in which one can reason about the behaviour of programs (as pointed out for instance by Reus [40]).

Purely syntactic arguments such as subject reduction suffice for proving the soundness of traditional type systems. However, once such type systems are turned into powerful

1998 ACM Subject Classification: D.3.1, F.3.2.

Key words and phrases: Formal calculi, objects, type systems, programming language semantics.

* A preliminary version of this paper was presented at the International Workshop on Foundations of Object-Oriented Languages (FOOL'08), 13 January 2008, San Francisco, California.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-5 (4:2) 2009

specification languages, like the logic of objects of Abadi and Leino [4] or the hybrid type system of Flanagan *et al.* [24], purely syntactic arguments seem no longer appropriate. The meaning of assertions is no longer obvious, since they have to describe the code on the heap. We believe that specifications of program behaviour should have a meaning independent of the particular proof system on which syntactic preservation proofs rely, as also argued by Benton [13] and by Reus and Schwinghammer [41].

In the case of specifications one would ideally prove soundness with respect to a semantic model that makes a clear distinction between semantic validity and derivability using the syntactic rules. However, building such semantic models is challenging, and there is currently no fully satisfactory semantic account of the imperative object calculus:

Denotational semantics: Domain-theoretic models have been employed in proving the soundness of the logic of Abadi and Leino [41, 42]. However, the existing techniques fall short of providing convincing models of *typed* objects: Reus and Streicher [42] consider an untyped semantics, and the model presented by Reus and Schwinghammer [41] handles neither second-order types, nor subtyping in depth. Due to the dynamically-allocated higher-order store present in the imperative object calculus, the models rely on techniques for recursively defined domains in functor categories [31, 36]. This makes them complex, and establishing properties even for specific programs often requires a substantial effort.

Equational reasoning: Gordon *et al.* [25] develop reasoning principles for establishing the contextual equivalence of untyped objects, and apply them to prove correctness of a compiler optimization. Jeffrey and Rathke [29] consider a concurrent variant of the calculus and characterize may-testing equivalence in terms of the trace sets generated by a labeled transition system. In both cases the semantics is limited to equational reasoning, *i.e.*, establishing contextual equivalences between programs. In theory, this can be used to verify a program by showing it equivalent to one that is trivially correct and acts as a specification. However, this can be more cumbersome in practice than using program logics, the established formalism for specifying and proving the correctness of programs.

Translations: Abadi *et al.* [3] give an adequate encoding of the imperative object calculus into a lambda calculus with records, references, recursive and existential types and subtyping. Together with an interpretation of this target language, an adequate model for the imperative object calculus could, in principle, be obtained. However, we are not aware of any worked-out adequate domain-theoretic models for general references and impredicative second-order types. Even if such a model was given, it would still be preferable to have a self-contained semantics for the object calculus, without the added complexity of the (non-trivial) translation.

A solution to the problem of finding adequate models of objects could be the step-indexed semantic models of types, introduced by Appel and McAllester [10] as an alternative to subject reduction proofs. Such models are based directly on the operational semantics, and are more easy to construct than the existing domain-theoretic models. The types are simply interpreted as sets of syntactic values indexed by a number of computation steps. Intuitively, a term belongs to a certain type if it behaves like an element of that type for any number of steps. Every type is built as a sequence of increasingly accurate semantic approximations, which allows one to easily deal with recursion. Type safety is an immediate consequence of this interpretation of types, and the semantic counterparts of the usual typing rules are proved as independent lemmas, either directly or by induction on the index. Ahmed

et al. [6, 9] successfully applied this generic technique to a lambda calculus with general references, impredicative polymorphism and recursive types.

In this paper we further extend the semantics of Ahmed *et al.* with object types and subtyping, and we use the resulting interpretation to prove the soundness of an expressive type system for the imperative object calculus. The main contribution of our work is the novel semantics of object types. We extend this semantics in two orthogonal ways. First, we adapt it to self types, *i.e.*, recursive object types that validate the usual subtyping rules as well as strong typing rules with structural assumptions. Second, we study a natural generalization of object types that results in simpler and more expressive typing rules.

Even though in this paper we are concerned with the safety of a type system, the step-indexing technique is not restricted to types, and has already been used for equational reasoning [5, 7, 10] and for proving the soundness of Hoare-style program logics of low-level languages [13, 14]. We expect therefore that it will eventually become possible to use a step-indexed model to prove the soundness of more expressive program logics for the imperative object calculus.

Outline. The next section introduces the syntax, operational semantics, and type system that we consider for the imperative object calculus. In Section 3 we present a step-indexed semantics for this calculus. In particular, we define the interpretations of types and establish their semantic properties. In Section 4 these properties are used to prove the soundness of the type system. Section 5 studies self types, while Section 6 discusses a natural generalization of object types. Section 7 gives a comparison to related work and Section 8 concludes. The Appendix presents the proofs of the most interesting typing and subtyping lemmas for object types, while an earlier technical report contains additional proofs [28].

2. The Imperative Object Calculus

We recall the syntax of the imperative object calculus with recursive and second-order types, and introduce a small-step operational semantics for this calculus that is equivalent to the big-step semantics given by Abadi and Cardelli [2].

2.1. **Syntax.** Let *Var*, *TVar* and *Meth* be pairwise disjoint, countably infinite sets of *variables*, *type variables* and *method names*, respectively. Let x, y range over *Var*, X, Y range over *TVar*, and let m range over *Meth*. Figure 1 defines the syntax of the types and terms of the imperative object calculus.

Objects are unordered collections of named methods, written as $[\mathrm{m}_d{=}\varsigma(x_d{:}A)b_d]_{d\in D}$. In a method $\mathrm{m} = \varsigma(x{:}A)b$, ς is a binder that binds the 'self' argument x in the method body b. The self argument can be used inside the method body for invoking the methods of the containing object. Methods with arguments other than self can be obtained by having a procedure as the method body. The methods of an object can be invoked or updated, but no new methods can be added, and the existing methods cannot be deleted. The type of objects with methods named m_d that return results of type A_d, for d in some set D, is written as $[\mathrm{m}_d :_{\nu_d} A_d]_{d\in D}$, where $\nu \in \{\circ, +, -\}$ is a *variance annotation* that indicates if the method is considered *invoke-only* $(+)$, *update-only* $(-)$, or if it may be used without restriction $(\circ)$.

While procedural abstractions are sometimes defined in the imperative object calculus using an additional let construct, we include them as primitives. We write procedures

$$
\begin{array}{rll}
A, B, C ::= & X \mid \mathit{Top} \mid \mathit{Bot} \mid A \to B & \text{(type expressions)} \\
\mid & [\mathrm{m}_d :_{\nu_d} A_d]_{d \in D} \mid \mu(X)A & \\
\mid & \forall (X \leqslant A)B \mid \exists (X \leqslant A)B & \\
\nu ::= & \circ \mid + \mid - & \text{(variance annotations)} \\
a, b ::= & x & \text{(variable)} \\
\mid & [\mathrm{m}_d = \varsigma(x_d{:}A)b_d]_{d \in D} & \text{(object creation)} \\
\mid & a.\mathrm{m} & \text{(method invocation)} \\
\mid & a.\mathrm{m} := \varsigma(x{:}A)b & \text{(method update)} \\
\mid & \text{clone } a & \text{(shallow copy)} \\
\mid & \lambda(x{:}A)b & \text{(procedure)} \\
\mid & a\ b & \text{(application)} \\
\mid & \mathrm{fold}_A\, b & \text{(recursive folding)} \\
\mid & \mathrm{unfold}_A\, b & \text{(recursive unfolding)} \\
\mid & \Lambda(X \leqslant A)b & \text{(type abstraction)} \\
\mid & a[A] & \text{(type application)} \\
\mid & \text{pack } X \leqslant A = C \text{ in } a : B & \text{(existential package)} \\
\mid & \text{open } a \text{ as } X \leqslant A, x{:}B \text{ in } b : C & \text{(package opening)}
\end{array}
$$

Figure 1: Syntax of types and terms

with type $A \to B$ as $\lambda(x{:}A)b$ and applications as $a\ b$, respectively. We use fold_A and unfold_A to denote the isomorphism between a recursive type $\mu(X)B$ and its unfolding $\{\!\{X \mapsto \mu(X)B\}\!\}(B)$. Finally, we consider bounded universal and existential types $\forall(X \leqslant A)B$ and $\exists(X \leqslant A)B$ along with their introduction and elimination forms [21].

The set of free variables of a term a is denoted by $\mathit{fv}(a)$, and similarly the free type variables in a type A by $\mathit{fv}(A)$. We identify types and terms up to the consistent renaming of bound variables. We use $\{\!\{t \mapsto r\}\!\}$ to denote the singleton map that maps t to r. For a finite map σ from variables to terms, $\sigma(a)$ denotes the result of capture-avoiding substitution of all $x \in \mathit{fv}(a) \cap \mathit{dom}(\sigma)$ by $\sigma(x)$. The same notation is used for the substitution of type variables. Generally, for any function f, the notation $f[t := r]$ denotes the function that maps t to r, and otherwise agrees with f.

2.2. **Operational Semantics.** Let *Loc* be a countably infinite set of *heap locations* ranged over by l. We extend the set of terms by run-time representations of objects $\{\mathrm{m}_d{=}l_d\}_{d \in D}$, associating heap locations to a set of method names. *Values* are given by the grammar:

$$v \in \mathit{Val} ::= \{\mathrm{m}_d{=}l_d\}_{d \in D} \mid \lambda(x{:}A)b \mid \mathrm{fold}_A\, v \mid \Lambda(X \leqslant A)b \mid \text{pack } X \leqslant A = C \text{ in } v : B$$

Apart from run-time objects, values consist of procedures, values of recursive type, type abstractions and existential packages as in the call-by-value lambda calculus. We often only consider terms and values without free variables, and denote the set of these *closed terms*

$$\mathcal{E}[\cdot] ::= [\cdot] \mid \mathcal{E}.\mathrm{m} \mid \mathcal{E}.\mathrm{m} \coloneqq \varsigma(x{:}A)b \mid \text{clone } \mathcal{E} \mid \mathcal{E}\ b \mid v\ \mathcal{E} \mid \text{fold}_A\, \mathcal{E} \mid \text{unfold}_A\, \mathcal{E} \mid \mathcal{E}[A] \mid \text{pack } X{\leqslant}A = C \text{ in } \mathcal{E} : B \mid \text{open } \mathcal{E} \text{ as } X{\leqslant}A, x{:}B \text{ in } b : C$$

Figure 2: Evaluation contexts

$$\begin{array}{lll}
\text{(Red-Obj)} & \langle h, [\mathrm{m}_d{=}\varsigma(x_d{:}A)b_d]_{d\in D}\rangle \to \langle h\,[l_d \coloneqq \lambda(x_d{:}A)b_d]_{d\in D}\,, \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle & \text{where } \forall d \in D.\ l_d \notin \mathit{dom}(h) \\
\text{(Red-Inv)} & \langle h, \{\mathrm{m}_d{=}l_d\}_{d\in D}.\mathrm{m}_e\rangle \to \langle h, h(l_e)\ \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle, \text{ if } e{\in}D & \\
\text{(Red-Upd)} & \langle h, \{\mathrm{m}_d{=}l_d\}_{d\in D}.\mathrm{m}_e \coloneqq \varsigma(x{:}A)b\rangle \to \langle h\,[l_e \coloneqq \lambda(x{:}A)b]\,, \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle, \text{ if } e{\in}D & \\
\text{(Red-Clone)} & \langle h, \text{clone } \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle \to \langle h\,[l'_d \coloneqq h(l_d)]_{d\in D}\,, \{\mathrm{m}_d{=}l'_d\}_{d\in D}\rangle & \text{where } \forall d \in D.\ l'_d \notin \mathit{dom}(h) \\
\text{(Red-Beta)} & \langle h, (\lambda(x{:}A)b)\ v\rangle \to \langle h, \{\!\{x \mapsto v\}\!\}(b)\rangle & \\
\text{(Red-Unfold)} & \langle h, \text{unfold}_A\,(\text{fold}_B\, v)\rangle \to \langle h, v\rangle & \\
\text{(Red-TBeta)} & \langle h, (\Lambda(X{\leqslant}A)b)[B]\rangle \to \langle h, \{\!\{X \mapsto B\}\!\}(b)\rangle & \\
\text{(Red-Open)} & \langle h, \text{open } v \text{ as } X{\leqslant}A, x{:}B \text{ in } b : C\rangle \to \langle h, \{\!\{x \mapsto v', X \mapsto C'\}\!\}(b)\rangle & \text{where } v \equiv \text{pack } X'{\leqslant}A' = C' \text{ in } v' : B'
\end{array}$$

Figure 3: One-step reduction relation

and *closed values* by *CTerm* and *CVal*, respectively. A *program* is a closed term that does not contain any locations, and we denote the set of all programs by *Prog*. A *heap* h is a finite map from *Loc* to *CVal*[1], and we write *Heap* for the set of all heaps.

Figure 2 defines the set of *evaluation contexts*, formalizing a left-to-right, call-by-value strategy. We write $\mathcal{E}[a]$ for the term obtained by plugging a into the hole $[\cdot]$ of $\mathcal{E}$. The one-step reduction relation $\to$ is defined as the least relation on *configurations* $\langle h, a\rangle \in \mathit{Heap} \times \mathit{CTerm}$ generated by the rules in Figure 3 and closed under the following context rule:

$$\langle h, a\rangle \to \langle h', a'\rangle \implies \langle h, \mathcal{E}[a]\rangle \to \langle h', \mathcal{E}[a']\rangle \qquad \text{(Red-Ctx)}$$

The methods are actually stored in the heap as procedures. Object construction allocates new heap storage for these procedures and returns a record of references to them (Red-Obj). Upon method invocation the corresponding stored procedure is retrieved from the heap and applied to the enclosing object (Red-Inv). The self parameter is thus passed just like any other procedure argument. Identifying methods and procedures makes the 'self-application' semantics of method invocation explicit, while technically it allows us to use the step-indexed model of Ahmed *et al.* [6, 9] with only few modifications.

While variables are immutable identifiers, methods can be updated destructively. Such updates only modify the heap and leave the run-time object unchanged (Red-Upd). Object

[1]In fact, for the purpose of modelling the imperative object calculus it would suffice to regard procedures as the only kind of storable value.

Subtyping $\boxed{\Gamma \vdash A \leqslant B}$

$$(\textsc{SubRefl})\ \frac{\Gamma \vdash A}{\Gamma \vdash A \leqslant A} \qquad (\textsc{SubTrans})\ \frac{\Gamma \vdash A \leqslant A' \quad \Gamma \vdash A' \leqslant B}{\Gamma \vdash A \leqslant B}$$

$$(\textsc{SubTop})\ \frac{\Gamma \vdash A}{\Gamma \vdash A \leqslant \mathit{Top}} \qquad (\textsc{SubBot})\ \frac{\Gamma \vdash A}{\Gamma \vdash \mathit{Bot} \leqslant A} \qquad (\textsc{SubVar})\ \frac{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash \diamond}{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash X \leqslant A}$$

$$(\textsc{SubProc})\ \frac{\Gamma \vdash A' \leqslant A \quad \Gamma \vdash B \leqslant B'}{\Gamma \vdash A \to B \leqslant A' \to B'}$$

$$(\textsc{SubObj})\ \frac{\begin{array}{r} E \subseteq D \quad \forall e{\in}E.\ (\nu_e \in \{+, \circ\} \Rightarrow \Gamma \vdash A_e \leqslant B_e) \\ \wedge\ (\nu_e \in \{-, \circ\} \Rightarrow \Gamma \vdash B_e \leqslant A_e) \end{array}}{\Gamma \vdash [\mathrm{m}_d :_{\nu_d} A_d]_{d \in D} \leqslant [\mathrm{m}_e :_{\nu_e} B_e]_{e \in E}}$$

$$(\textsc{SubObjVar})\ \frac{\forall d \in D.\ \nu_d = \circ\ \vee\ \nu_d = \nu'_d}{\Gamma \vdash [\mathrm{m}_d :_{\nu_d} A_d]_{d \in D} \leqslant [\mathrm{m}_d :_{\nu'_d} A_d]_{d \in D}}$$

$$(\textsc{SubRec})\ \frac{\Gamma \vdash \mu(X)A \quad \Gamma \vdash \mu(Y)B \quad \Gamma, Y{\leqslant}\mathit{Top}, X{\leqslant}Y \vdash A \leqslant B}{\Gamma \vdash \mu(X)A \leqslant \mu(Y)B}$$

$$(\textsc{SubUniv})\ \frac{\Gamma \vdash A' \leqslant A \quad \Gamma, X{\leqslant}A' \vdash B \leqslant B'}{\Gamma \vdash \forall(X{\leqslant}A)B \leqslant \forall(X{\leqslant}A')B'}$$

$$(\textsc{SubExist})\ \frac{\Gamma \vdash A \leqslant A' \quad \Gamma, X{\leqslant}A \vdash B \leqslant B'}{\Gamma \vdash \exists(X{\leqslant}A)B \leqslant \exists(X{\leqslant}A')B'}$$

Figure 4: Subtyping

cloning generates a shallow copy of an object in the heap (Red-Clone). The last four rules in Figure 3 are as in the lambda calculus.

For $k \in \mathbb{N}$, $\rightarrow^k$ denotes the k-step reduction relation. We write $\langle h, a\rangle \nrightarrow$ if the configuration $\langle h, a\rangle$ is irreducible (*i.e.*, there exists no configuration $\langle h', a'\rangle$ such that $\langle h, a\rangle \rightarrow \langle h', a'\rangle$).

Note that reduction is not deterministic, due to the arbitrarily chosen fresh locations in (Red-Obj) and (Red-Clone). However, we still have that there is always at most one, uniquely determined redex. This has the important consequence that the reduction order is fixed. For example, if there is a reduction sequence beginning with a method invocation and ending in an irreducible configuration: $\langle h_1, a.\mathrm{m}\rangle \rightarrow^k \langle h_2, b\rangle \nrightarrow$, then this sequence can be split into

$$\langle h_1, a.\mathrm{m}\rangle \rightarrow^i \langle h'_1, a'.\mathrm{m}\rangle \rightarrow^{k-i} \langle h_2, b\rangle$$

where $\langle h_1, a\rangle \rightarrow^i \langle h'_1, a'\rangle \nrightarrow$ for some $i \geq 0$. Similar decompositions into subsequences hold for reductions starting from the other term forms.

It is easy to see that the operational semantics is independent of the type annotations inside terms. Also the semantic types that we define in Section 3 will not depend on the syntactic type expressions in the terms. In order to reduce the notational overhead and to prevent confusion between the syntax and semantics of types we will omit type annotations

Subsumption and axioms $\boxed{\Gamma \vdash a : A}$

$$(\textsc{Sub})\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash A \leqslant B}{\Gamma \vdash a : B} \qquad (\textsc{Var})\ \frac{\Gamma_1, x{:}A, \Gamma_2 \vdash \diamond}{\Gamma_1, x{:}A, \Gamma_2 \vdash x : A}$$

Procedure types

$$(\textsc{Lam})\ \frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x{:}A)b : A \to B} \qquad (\textsc{App})\ \frac{\Gamma \vdash a : B \to A \quad \Gamma \vdash b : B}{\Gamma \vdash a\ b : A}$$

Object types (where $A \equiv [\mathrm{m}_d :_{\nu_d} A_d]_{d \in D}$)

$$(\textsc{Obj})\ \frac{\forall d{\in}D.\ \Gamma, x_d{:}A \vdash b_d : A_d}{\Gamma \vdash [\mathrm{m}_d{=}\varsigma(x_d{:}A)b_d]_{d \in D} : A} \qquad (\textsc{Clone})\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{clone}\ a : A}$$

$$(\textsc{Inv})\ \frac{\Gamma \vdash a : A \quad e \in D \quad \nu_e \in \{+, \circ\}}{\Gamma \vdash a.\mathrm{m}_e : A_e}$$

$$(\textsc{Upd})\ \frac{\Gamma \vdash a : A \quad e \in D \quad \Gamma, x{:}A \vdash b : A_e \quad \nu_e \in \{-, \circ\}}{\Gamma \vdash a.\mathrm{m}_e \coloneqq \varsigma(x{:}A)b : A}$$

Recursive types

$$(\textsc{Unfold})\ \frac{\Gamma \vdash a : \mu(X)A}{\Gamma \vdash \mathrm{unfold}_{\mu(X)A}\, a : \{\!\{X \mapsto \mu(X)A\}\!\}(A)}$$

$$(\textsc{Fold})\ \frac{\Gamma \vdash a : \{\!\{X \mapsto \mu(X)A\}\!\}(A)}{\Gamma \vdash \mathrm{fold}_{\mu(X)A}\, a : \mu(X)A}$$

Bounded quantified types

$$(\textsc{TAbs})\ \frac{\Gamma, X{\leqslant}A \vdash b : B}{\Gamma \vdash \Lambda(X{\leqslant}A)b : \forall(X{\leqslant}A)B} \qquad (\textsc{TApp})\ \frac{\Gamma \vdash a : \forall(X{\leqslant}A)B \quad \Gamma \vdash A' \leqslant A}{\Gamma \vdash a[A'] : \{\!\{X \mapsto A'\}\!\}(B)}$$

$$(\textsc{Pack})\ \frac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash \{\!\{X \mapsto C\}\!\}(a) : \{\!\{X \mapsto C\}\!\}(B)}{\Gamma \vdash (\mathrm{pack}\ X{\leqslant}A = C\ \mathrm{in}\ a{:}B) : \exists(X{\leqslant}A)B}$$

$$(\textsc{Open})\ \frac{\Gamma \vdash a : \exists(X{\leqslant}A)B \quad \Gamma \vdash C \quad \Gamma, X{\leqslant}A, x{:}B \vdash b : C}{\Gamma \vdash (\mathrm{open}\ a\ \mathrm{as}\ X{\leqslant}A, x{:}B\ \mathrm{in}\ b{:}C) : C}$$

Figure 5: Typing of terms

when presenting the step-indexed semantics. For example, instead of the type application $a[A]$ we will merely write $a[]$.

2.3. **Type System.** The type system we consider features procedure, object, iso-recursive and (impredicative, bounded) quantified types, as well as subtyping, and corresponds to $\mathbf{FOb}_{<:\mu}$ from [2]. It is fairly standard and consists of four inductively defined typing judgments:

- $\Gamma \vdash \diamond$, describing *well-formed typing contexts*,
- $\Gamma \vdash A$, defining *well-formed types*,

- $\Gamma \vdash A \leqslant B$, for *subtyping* between well-formed types, and
- $\Gamma \vdash a : A$, for *typing terms*.

The typing context Γ is a list containing type bindings for the (term) variables $x{:}A$ and upper bounds for the type variables $X{\leqslant}A$. A typing context is well-formed if it does not contain duplicate bindings for (term or type) variables and all types appearing in it are well-formed. A type is well-formed with respect to a well-formed context Γ if all its type variables appear in Γ.

Figure 4 defines the subtyping relation. For the object types it allows subtyping in width: an object type with more methods is a subtype of an object type with fewer methods, as long as the types of the common methods agree. For the invoke-only ($+$) and update-only methods ($-$) in object types, covariant respectively contravariant subtyping in depth is allowed (SubObj). Furthermore, the unrestricted methods ($\circ$) can be regarded, by subtyping, as either invoke-only or update-only (SubObjVar). Since the annotations can be conveniently chosen at creation time (Obj) this brings much flexibility. As explained by Abadi and Cardelli [2], this allows us to distinguish in the type system between the invocations and updates done through the self argument, and the ones done from the outside. The main idea is to type an object creation with an object type where all methods are considered invariant, so that all invocations and updates through the self argument (internal) are allowed, but have to be type preserving. Then rules (Sub) and (SubObjVar) are applied and some of the methods can become invoke-only, some others update-only. This enables the subsequent weakening of the types of these methods using (SubObj). In effect, this allows for safe and flexible subtyping of methods, at the price of restricting update and invocation of the methods from the outside. Nevertheless, the internal updates and invocations remain unrestricted.

Figure 5 defines the typing relation. The applicability of the rules for method invocation (Inv), and for method update (Upd), depends on the variance annotation. Also notice that only type-preserving updates are allowed in (Upd). Finally, it is important to note that we do not give types to heap locations, since the type system is only used to check programs, and programs do not contain locations. In contrast, a proof of type safety using the preservation and progress properties would require the syntactic judgement to also depend on a heap typing since partially evaluated terms would also need to be typed.

3. A Step-indexed Semantics of Objects

Modelling higher-order store is necessarily more involved than the treatment of first-order storage since the semantic domains become mutually recursive. Recall that heaps store values that may be procedures. These in turn can be modeled as functions that take a value and the initial heap as input, and return a value and the possibly modified heap upon termination. This suggests the following semantic domains for values and heaps, respectively:

$$\begin{aligned} D_{Val} &= (D_{Heaps} \times D_{Val} \rightharpoonup D_{Heaps} \times D_{Val}) + \dots \\ D_{Heaps} &= Loc \rightharpoonup_{fin} D_{Val} \end{aligned} \tag{3.1}$$

A simple cardinality argument shows that there are no set-theoretic solutions (*i.e.*, where $D \rightharpoonup E$ denotes the set of all partial functions from D to E) satisfying the equations in (3.1). A possible solution is to use a domain-theoretic approach, as done for the imperative object calculus by Reus and Streicher [42], building on earlier work by Kamin and Reddy [30].

In a model of a typed calculus one also wants to interpret the types. But naively taking a collection *Type* of subsets $\tau \subseteq D_{Val}$ as interpretations of syntactic types does not work, since values generally depend on the heap and a typed model should guarantee that all heap access operations are type-correct. We are led to the following approach: first, in order to ensure that updates are type-preserving, we also consider *heap typings*. Heap typings are partial maps $\Psi \in \mathit{HeapTyping} = \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Type}$ that track the set of values that may be stored in each heap location. Second, the collection of types is refined to take heap typings into account: a type will now consist of values paired with heap typings that describe the necessary requirements on heaps. These ideas suggest that we take

$$\begin{aligned} \mathit{Type} &= \mathcal{P}(\mathit{HeapTyping} \times D_{Val}) \\ \mathit{HeapTyping} &= \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Type} \end{aligned} \tag{3.2}$$

Again, a cardinality argument shows the impossibility of defining these sets.

A final obstacle to modelling the object calculus, albeit independent of the higher-order nature of heaps, is due to dynamic allocation in the heap. This results in heap typings that may *vary* in the course of a computation, reflecting the changing 'shape' of the heap. However, as is the case for many high-level languages, the object calculus is well-behaved in this respect:

- inside the language, there is no possibility of deallocating heap locations; and
- only weak (*i.e.*, type-preserving) updates are allowed.

As a consequence, *extensions* are the only changes that need to be considered for heap typings. Intuitively, values that rely on heaps with typing Ψ will also be type-correct for extended heaps, with an extended heap typing $\Psi' \sqsupseteq \Psi$. For this reason, semantic models of dynamic allocation typically lend themselves to a Kripke-style presentation, where all semantic entities are indexed by *possible worlds* drawn from the set of heap typings, partially (pre-) ordered by heap typing extension [31, 33, 34, 37, 39].

Rather than trying to extend the already complex domain-theoretic models to heap typings and dynamic allocation, we will use the step-indexing technique. Since this technique is based directly on the operational semantics, it provides an alternative that has less mathematical overhead. In particular, there is no need to find semantic domains satisfying (3.1); we can simply have D_{Val} be the set of closed values and use syntactic procedures in place of set-theoretic functions. Moreover, it is relatively easy to also model impredicative second-order types in the step-indexed model of Ahmed *et al.* [6, 9], which is crucial for the interpretation of object types we develop below. Although recently there has been progress in finding domain-theoretic models of languages that combine references and polymorphic types [15, 16, 17], the constructions are more involved.

The circularity in (3.2) is resolved by considering a stratification based on a notion of 'k-step execution safety'. The central idea is that a term has type τ with approximation k if this assumption cannot be proved wrong (in the sense of reaching a stuck state) in any context by executing fewer than k steps. The key insight for constructing the sets satisfying (3.2) is that all operations on the heap consume one step. Thus, in order to determine whether a pair $\langle \Psi, v \rangle$, where Ψ is a heap typing and v a value, belongs to a type τ with approximation k it is sufficient to know the types of the stored values on which v relies (as recorded by Ψ) only up to level $k-1$. The true meaning of types and heap typings is then obtained by taking the limit over all such approximations.

For instance, if a heap typing Ψ asserts that a *Bool*-returning procedure is stored at location l, *i.e.*, $\Psi(l) = [\mathrm{m}{:}\mathit{Bool}] \to \mathit{Bool}$, then it is certainly not safe to assume that the

pair $\langle \Psi, \lambda(y)\{\mathrm{m}{=}l\}.\mathrm{m}\rangle$ belongs to the type of *Int*-returning procedures. However, it is not possible to contradict this assumption by taking only two reduction steps: the first step is consumed by the beta reduction, the second one by the method selection $\{\mathrm{m}{=}l\}.\mathrm{m}$ in the procedure body, which involves a heap access. In this case, there are no steps left to observe that the result of the computation is a boolean rather than an integer. Consequently, the value $\lambda(y)\{\mathrm{m}{=}l\}.\mathrm{m}$ is in the type of *Int*-returning procedures for two computation steps, even though it does not actually return an integer. One can of course distinguish such 'false positives' by taking more reduction steps.

The preceding considerations are now formalized, building on the model originally developed by Ahmed *et al.* for an ML-like language with general references and impredicative second-order types [6, 9]. Apart from some notational differences, the definitions in Section 3.1 are the same as in [6]. Section 3.2 adds subtyping, while Section 3.3 deals with procedure types, and Section 3.4 revisits reference types. The semantics of object types is presented in Section 3.5 and constitutes the main contribution of this paper. We further deviate from [6] by adding bounds to the second-order types in Section 3.6, and by using iso-recursive instead of equi-recursive types in Section 3.7.

3.1. **The Semantic Model.** To make the (circular) definition of types and heap typings from (3.2) work, the step-indexed semantics considers triples with an additional natural number component, representing the step index, rather than just pairs. First, we inductively define two families $(\mathit{PreType}_k)_{k\in\mathbb{N}}$ of *pre-types*, and $(\mathit{HeapPreTyping}_k)_{k\in\mathbb{N}}$ of *heap pre-typings*, by

$$\begin{aligned}
\tau \in \mathit{PreType}_0 &\;\Leftrightarrow\; \tau = \emptyset \\
\tau \in \mathit{PreType}_{k+1} &\;\Leftrightarrow\; \tau \in \mathcal{P}(\mathbb{N} \times (\textstyle\bigcup_{j\le k} \mathit{HeapPreTyping}_j) \times \mathit{CVal}) \\
&\qquad \wedge\; \forall \langle j, \Psi, v\rangle \in \tau.\; j \le k \;\wedge\; \Psi \in \mathit{HeapPreTyping}_j
\end{aligned}$$

where $\mathit{HeapPreTyping}_k = \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{PreType}_k$. That is, each $\tau \in \mathit{PreType}_k$ is a set of triples $\langle j, \Psi, v\rangle$ where the set $\mathit{HeapPreTyping}_j$ from which the heap pre-typing Ψ is drawn depends on the index $j < k$. Clearly $\mathit{PreType}_k \subseteq \mathit{PreType}_{k+1}$ and thus $\mathit{HeapPreTyping}_k \subseteq \mathit{HeapPreTyping}_{k+1}$ for all k. Now it is possible to set

$$\begin{aligned}
\tau \in \mathit{PreType} &\;\Leftrightarrow\; \tau \in \mathcal{P}(\mathbb{N} \times (\textstyle\bigcup_{j} \mathit{HeapPreTyping}_j) \times \mathit{CVal}) \\
&\qquad \wedge\; \forall \langle j, \Psi, v\rangle \in \tau.\; \Psi \in \mathit{HeapPreTyping}_j
\end{aligned}$$

We call the elements of this set *pre*-types, rather than types, since there will be a further condition that proper types must satisfy (this is done in Definition 3.4 below). From now on, when writing $\langle k, \Psi, v\rangle$, we always implicitly assume that $\Psi \in \mathit{HeapPreTyping}_k$. By *HeapPreTyping* we denote the set $\mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{PreType}$ of finite maps into pre-types.

Each pre-type τ is a union of sets $\tau_k \in \mathit{PreType}_k$ where the index appearing in elements of τ_k is bounded by k. This is made explicit by the following notion of semantic approximation and the stratification invariant below.

Definition 3.1 (Semantic approximation)**.** For any pre-type τ we call $\lfloor\tau\rfloor_k$ the *k-th approximation of τ* and define it as the subset containing all elements of τ that have an index strictly less than k: $\lfloor\tau\rfloor_k = \{\langle j, \Psi, v\rangle \in \tau \mid j < k\}$. This definition is lifted pointwise to the (partial) functions in *HeapPreTyping*: $\lfloor\Psi\rfloor_k = \lambda l \in \mathit{dom}(\Psi).\ \lfloor\Psi(l)\rfloor_k$.

Proposition 3.2 (Stratification)**.** *For all $\tau \in \mathit{PreType}$ and $k \in \mathbb{N}$, $\lfloor\tau\rfloor_k \in \mathit{PreType}_k$. Moreover, $\tau = \bigcup_k \lfloor\tau\rfloor_k$.* □

So in particular, if $\langle k, \Psi, v\rangle \in \tau$ and $l \in dom(\Psi)$ then $\Psi(l) \in PreType_j$ for some $j \leq k$. This is captured by the following 'stratification invariant', which will be satisfied by all the constructions on (pre-) types, and which ensures the well-foundedness of the whole construction:

> **Stratification invariant.** For all pre-types τ, $\lfloor\tau\rfloor_{k+1}$ cannot depend on any pre-type beyond approximation k.

As indicated above, in order to take dynamic allocation into account we consider a possible worlds model. Intuitively we think of a pair (k, Ψ) as describing the *state* of a heap h, where Ψ lists locations in h that are guaranteed to be allocated, and contains the types of the stored values up to approximation k. In the course of a computation, there are three different situations where the heap state changes:

- New objects are allocated on the heap, which is reflected by a heap pre-typing Ψ' with additional locations compared to Ψ. This operation does not affect any of the previously stored objects, so Ψ' will be an extension of Ψ.
- The program executes for $k - j$ steps, for some $j \leq k$, without accessing the heap. This is reflected by a heap state $(j, \lfloor\Psi\rfloor_j)$ that 'forgets' that we have a more precise approximation, and guarantees that the heap is safe only for j execution steps.
- The heap is updated, but in such a way that all typing guarantees of Ψ are preserved. Thus updates will be reflected by an information forgetting extension, as in the previous case. However, because of the step taken by the update itself, in this case we necessarily have that $j < k$.

The following definition of state extension captures these possible evolutions of a state.

Definition 3.3 (State extension)**.** *State extension* $\sqsubseteq$ is the relation on $\mathbb{N} \times HeapPreTyping$ defined by

$$\begin{aligned}(k, \Psi) \sqsubseteq (j, \Psi') \;\Leftrightarrow\; & j \leq k \;\wedge\; dom(\Psi) \subseteq dom(\Psi') \\ & \wedge\; \forall l \in dom(\Psi).\; \left\lfloor\Psi'\right\rfloor_j(l) = \lfloor\Psi\rfloor_j(l)\end{aligned}$$

The step-indexing technique relies on the approximation of the 'true' set of values that constitute a type, by all those values that behave accordingly unless a certain number of computation steps are taken. Limiting the number of available steps, we will only be able to make fewer distinctions. Moreover, if for instance a procedure relies on locations in the heap as described by a state (k, Ψ), we can safely apply the procedure after further allocations. In fact, if we are only interested in safely executing the procedure for $j < k$ steps, a heap described by state $(j, \lfloor\Psi\rfloor_j)$ will suffice. These conditions are captured precisely by state extension, so we require our semantic types to be closed under state extension:

Definition 3.4 (Semantic types and heap typings)**.** The set *Type* of *semantic types* is the subset of *PreType* defined by

$$\begin{aligned}\tau \in Type \;\Leftrightarrow\; & \forall k, j \geq 0.\, \forall \Psi, \Psi'.\, \forall v \in CVal. \\ & (k, \Psi) \sqsubseteq (j, \Psi') \;\wedge\; \langle k, \Psi, v\rangle \in \tau \;\Rightarrow\; \langle j, \Psi', v\rangle \in \tau\end{aligned}$$

We also define the set $HeapTyping = Loc \rightharpoonup_{fin} Type$ of *heap typings*, ranged over by Ψ in the following, as the subset of heap pre-typings that map to semantic types.

As explained by Ahmed [6], this structure may be viewed as an instance of Kripke models of intuitionistic logic where states are the possible worlds, state extension is the

reachability relation between worlds, and where closure under state extension corresponds to Kripke monotonicity.

Next we define when a particular heap h conforms to the requirements expressed by a heap typing Ψ. This is done with respect to an approximation index.

Definition 3.5 (Well-typed heap)**.** A heap h is *well-typed* with respect to Ψ with approximation k, written as $h :_k \Psi$, if $dom(\Psi) \subseteq dom(h)$ and

$$\forall j < k.\ \forall l \in dom(\Psi).\ \langle j, \lfloor \Psi \rfloor_j, h(l) \rangle \in \Psi(l)$$

Semantic types only contain values, but we also need to associate types with terms that are not values. We do this in two steps, first for closed terms, then for arbitrary ones. A closed term has a certain type to approximation k with respect to some heap typing Ψ, if in all heaps that are well-typed with respect to Ψ the term behaves like an element of the type for k computation steps. In general, before reducing to a value the term will execute for j steps, and possibly allocate some new heap locations in doing so. The state describing the final heap will therefore be an extension of the state describing the initial heap, and it only needs to be safe for the remaining $k - j$ steps. Similarly, the final value needs to be in the original type only for another $k - j$ steps. The next definition makes this precise.

Definition 3.6 (Closed term has semantic type)**.** We say that a closed term a *has type* τ with respect to the state (k, Ψ), denoted as $a :_{k,\Psi} \tau$, if and only if

$$\begin{aligned}\forall j < k, h, h', b.\ (h :_k \Psi\ \wedge\ \langle h, a\rangle \rightarrow^j \langle h', b\rangle\ \wedge\ \langle h', b\rangle \not\rightarrow) \\ \Rightarrow \exists \Psi'.\ (k, \Psi) \sqsubseteq (k - j, \Psi')\ \wedge\ h' :_{k-j} \Psi'\ \wedge\ \langle k - j, \Psi', b\rangle \in \tau\end{aligned}$$

Even though the terms we evaluate are closed, when type-checking their subterms we also have to reason about open terms. Typing open terms is done with respect to a semantic type environment Σ that maps variables to semantic types. We reduce typing open terms to typing their closed instances obtained by substituting all free variables with appropriately typed, closed values. This is done by a value environment σ (a finite map from variables to closed values) that agrees with the type environment.

Definition 3.7 (Value environment agrees with type environment)**.** We say that *value environment* σ *agrees with semantic type environment* Σ*, with respect to the state* (k, Ψ), if $\forall x \in dom(\Sigma).\ \sigma(x) :_{k,\Psi} \Sigma(x)$. We denote this by $\sigma :_{k,\Psi} \Sigma$.

Definition 3.8 (Semantic typing judgement)**.** We say that a term a (possibly with free variables, but not containing locations), *has type* τ with respect to a semantic type environment Σ, written as $\Sigma \models a : \tau$, if after substituting well-typed values for the free variables of a, we obtain a closed term that has type τ for any number of computation steps. More precisely:

$$\Sigma \models a : \tau \quad \Leftrightarrow \quad fv(a) \subseteq dom(\Sigma) \quad \wedge\ \forall k \geq 0.\ \forall \Psi.\ \forall \sigma :_{k,\Psi} \Sigma.\ \sigma(a) :_{k,\Psi} \tau$$

By construction, the semantic typing judgment enforces that all terms that are typable with respect to it do not produce type errors when evaluated.

Definition 3.9 (Safe for k steps)**.** We call a configuration $\langle h, a\rangle$ *safe for* k *steps*, if the term a does not get stuck in less than k steps when evaluated in the heap h, *i.e.*, we define the set of all such configurations by

$$Safe_k = \{\langle h, a\rangle \mid \forall j < k.\ \forall h', b.\ \langle h, a\rangle \rightarrow^j \langle h', b\rangle\ \wedge\ \langle h', b\rangle \not\rightarrow \ \Rightarrow\ b \in Val\}$$

Definition 3.10 (Safety)**.** We call a configuration *safe* if it does not get stuck in any number of steps, and let $\mathit{Safe} = \bigcap_{k\in\mathbb{N}} \mathit{Safe}_k$.

Theorem 3.11 (Safety)**.** *For all programs a such that $\emptyset \models a : \tau$ and for all heaps h we have that $\langle h, a\rangle \in \mathit{Safe}$.*

Proof. One first easily shows that, if $a :_{k,\Psi} \tau$ and $h :_k \Psi$, then $\langle h, a\rangle \in \mathit{Safe}_k$. The theorem then follows by observing that any h is well-typed with respect to the empty heap typing, to any approximation k. □

This is much more direct than a subject reduction proof [46]. However, unlike with subject reduction, the validity of the typing rules still needs to be proved with respect to the semantics. We do this in two steps. In the remainder of this section we introduce the specific semantic interpretations of types, and prove that they satisfy certain semantic typing lemmas. These proofs are similar in spirit to proving the 'fundamental theorem' of Kripke logical relations [32]. Then, in Section 4 we prove the soundness of the rules of the initial type system with respect to these typing lemmas.

Even though the semantic typing lemmas are constructed so that they directly correspond to the rules of the original type system, there is a big difference between the two. While the semantic typing lemmas allow us to logically derive valid semantic judgments using other valid judgments as premises, the typing rules are just syntax that is used in the inductive definitions of the typing and subtyping relations.

3.2. **Subtyping.** Since types in the step-indexed interpretation are sets (satisfying some additional constraints), the natural subtyping relation is set inclusion. This subtyping relation forms a complete lattice on semantic types, where infima and suprema are given by set-theoretic intersections and unions, respectively. The least element is $\bot = \emptyset$, while the greatest is

$$\top = \{\langle j, \Psi, v\rangle \mid j \in \mathbb{N}, \Psi \in \mathit{HeapTyping}_j, v \in \mathit{CVal}\}.$$

Obviously $\bot$ and $\top$ satisfy both the stratification invariant (*i.e.*, they are pre-types) and the closure under state extension condition, so they are indeed semantic types.

We can easily show the standard subsumption property

Lemma 3.12 (Subsumption)**.** *If $\Sigma \models a : \alpha$ and $\alpha \subseteq \beta$ then $\Sigma \models a : \beta$.* □

While it is very easy to define subtyping in this way, the interaction between subtyping and the other features of the type system, in particular the object types, is far from trivial. This point will be discussed further in Section 3.5.

3.3. **Procedure Types.** Intuitively, a procedure has type $\alpha \to \beta$ for k computation steps if, when applied to any well-typed argument of type α, it produces a result that has type β for another $k-1$ steps. This is because the procedure application itself takes one computation step, and the only way to use a procedure is by applying it to some argument.

Additionally, we have to take into account that the procedure can also be applied after some computation steps that extend the heap. So, for every $j < k$ and for every heap typing Ψ' such that $(k, \Psi) \sqsubseteq (j, \Psi')$, when applying the procedure to a value in type α for j steps with respect to Ψ', the result must have type β for j steps with respect to Ψ'. This computational intuition nicely fits the possible worlds reading of procedure types as intuitionistic implication.

$$\Sigma[x := \alpha] \models b : \beta \implies \Sigma \models \lambda x.\, b : \alpha \to \beta \qquad \text{(SemLam)}$$

$$(\Sigma \models a : \beta \to \alpha \ \wedge\ \Sigma \models b : \beta \implies \Sigma \models a\ b : \alpha \qquad \text{(SemApp)}$$

$$\alpha' \subseteq \alpha \ \wedge\ \beta \subseteq \beta' \implies \alpha \to \beta \subseteq \alpha' \to \beta' \qquad \text{(SemSubProc)}$$

Figure 6: Typing lemmas: procedure types

Definition 3.13 (Procedure types)**.** If α and β are semantic types, then $\alpha \to \beta$ consists of those triples $\langle k, \Psi, \lambda x.\, b\rangle$ such that for all $j < k$, heap typings Ψ' and closed values v:

$$((k, \Psi) \sqsubseteq (j, \Psi') \ \wedge\ \langle j, \Psi', v\rangle \in \alpha) \ \Rightarrow\ \{\!\{x \mapsto v\}\!\}(b) :_{j,\Psi'} \beta$$

Proposition 3.14. *If α and β are semantic types, then $\alpha \to \beta$ is also a semantic type.* □

Figure 6 contains the semantic typing lemmas associated with procedure types. The procedure type constructor is of course contravariant in the argument type and covariant in the result type.

Lemma 3.15 (Procedure types)**.** *The three semantic typing lemmas shown in Figure 6 are valid implications.*

Proof sketch. The validity of (SemApp) and (SemLam) is proved in [6]. Verifying (SemSubProc) is simply a matter of unfolding the definitions. □

3.4. **Revisiting Reference Types.** While our calculus does not have references syntactically, we will use the model of references from [6, 9] in our construction underlying object types. In order to interpret the variance annotations in object types, we additionally introduce readable reference types and writable reference types, with covariant and contravariant subtyping, respectively [35, 43].

A heap typing associates with each allocated location the precise type that can be used when reading from it and writing to it. So all heap locations support both reading and writing at a certain type, and we do not have read-only or write-only locations. Intuitively, for the readable reference types and the writable ones the precise type of the locations is only partially known, so that without additional information only one of the two operations is safe at a meaningful type.

We first recall the definition of reference types from [6, 9].

Definition 3.16 (Reference types)**.** If τ is a semantic type then

$$\mathrm{ref}_\circ \tau = \{\langle k, \Psi, l\rangle \mid \lfloor \Psi(l)\rfloor_k = \lfloor \tau \rfloor_k\}$$

According to this definition, a location l has type $\mathrm{ref}_\circ\tau$ if the type associated with l by the heap typing Ψ is approximately τ. Semantic approximation is used to satisfy the stratification invariant, and is operationally justified by the fact that reading from a location or writing to it takes one computation step. So, l has type $\mathrm{ref}_\circ\tau$ for k steps if all values that are read from l or written to l have type τ for $k-1$ steps.

The readable reference type $\mathrm{ref}_+\tau$ is similar to $\mathrm{ref}_\circ\tau$, but poses less constraints on the heap typing Ψ: it only requires that $\Psi(l)$ is a subtype of τ, as before up to some approximation.

$$\alpha \subseteq \beta \implies \mathrm{ref}_+\alpha \subseteq \mathrm{ref}_+\beta \qquad \text{(SemSubCovRef)}$$
$$\beta \subseteq \alpha \implies \mathrm{ref}_-\alpha \subseteq \mathrm{ref}_-\beta \qquad \text{(SemSubConRef)}$$
$$\mathrm{ref}_\circ\alpha \subseteq \mathrm{ref}_\nu\alpha, \text{ where } \nu \in \{\circ, +, -\} \qquad \text{(SemSubVarRef)}$$

Figure 7: Subtyping reference types

Definition 3.17 (Readable reference types)**.** If τ is a semantic type then

$$\mathrm{ref}_+\tau = \{\langle k, \Psi, l\rangle \mid \lfloor\Psi\rfloor_k(l) \subseteq \lfloor\tau\rfloor_k\}$$

The value stored at location l also has type τ by subsumption, and therefore can be read and safely used as a value of type τ. However, the true type of location l is in general unknown, so writing any value to it could be unsafe (the true type of l might be the empty type $\bot$). Nevertheless, knowing that a location has type $\mathrm{ref}_+\tau$ does not mean that we cannot write to it: it simply means that we do not know the type of the values that can be written to it, so in the absence of further information no writing can be guaranteed to be type safe[2].

Dually, the type $\mathrm{ref}_-\tau$ of writable references contains all those locations l whose type associated by Ψ is a supertype of τ.

Definition 3.18 (Writable reference types)**.** If τ is a semantic type then

$$\mathrm{ref}_-\tau = \{\langle k, \Psi, l\rangle \mid \lfloor\tau\rfloor_k \subseteq \lfloor\Psi\rfloor_k(l)\}$$

We can safely write a value of type τ to a location of type $\mathrm{ref}_-\tau$, since this value also has the real type of location l by subsumption. However, the real type of such locations can be arbitrarily general. In particular it can be $\top$, the type of all values. Thus a location about which we only know that it has type $\mathrm{ref}_-\tau$ can only be read safely at type $\top$.

With these definitions in place, the usual reference type from Definition 3.16 can be recovered as the intersection of a readable and a writable reference type:

$$\mathrm{ref}_\circ\tau = \mathrm{ref}_+\tau \cap \mathrm{ref}_-\tau$$

Hence $\mathrm{ref}_+\tau$ and $\mathrm{ref}_-\tau$ are both supertypes of $\mathrm{ref}_\circ\tau$. It can also be easily shown that the readable reference type constructor is covariant, the writable reference type constructor is contravariant (Figure 7), while the usual reference types are obviously invariant. For a variance annotation $\nu \in \{\circ, +, -\}$ we use ref_ν to stand for the reference type constructor with this variance.

Note that, strictly speaking, the set $\mathrm{ref}_\nu\tau$ is not a semantic type since for our calculus locations are not values (although locations appear in object values $\{\mathrm{m}_d{=}l_d\}_{d\in D}$; see Section 2.2). In fact, the definition of object types (Definition 3.20 in the next section) will not depend on $\mathrm{ref}_\nu\tau$ being a semantic type. However, in order for the object type constructor to yield semantic types, it is crucial that $\mathrm{ref}_\nu\tau$ is closed under state extension.

Proposition 3.19. *If τ is a semantic type, then* $\mathrm{ref}_\nu\tau$ *is closed under state extension.* □

[2]This is conceptually different from the immutable reference types modeled in [6] using singleton types.

3.5. **Object Types.** Giving a semantics to object types is much more challenging than for the other types. The typing rules from Section 2 indicate why this is the case. First, an adequate interpretation of object types must permit subtyping both in width and in depth, taking the variance annotations into account. Second, in contrast to all the other types we consider that have just a single elimination rule, once constructed, objects support three different operations: invocation, update, and cloning. The definition of object types must ensure the consistent use of an object through all possible future operations. That is, all the requirements on which invocation, update or cloning rely must already be established at object creation time.

Before defining the object types, it is instructive to consider some simpler variants that do not fulfill all the requirements we have for object types.

Our decision to store methods in the heap as procedures, together with the 'self-application' semantics of method invocation (Red-Inv in Figure 3), suggest that object types are somewhat similar to recursive types of records of references holding procedures that take the enclosing record as argument:

$$[\mathrm{m}_d : \tau_d]_{d\in D} \stackrel{?}{=} \mu(\alpha).\{\mathrm{m}_d : \mathrm{ref}_\circ(\alpha \to \tau_d)\}_{d\in D}$$

However, the invariance of the reference type constructor blocks any form of subtyping, even in width. A look at typing rules for subtyping recursive types, such as Cardelli's Amber rule [20] (which appears as rule SubRec in Figure 4), suggests that the position of the recursion variable should be covariant. For instance, when attempting to establish the subtyping $[\mathrm{m}_1 : \tau_1, \mathrm{m}_2 : \tau_2] \subseteq [\mathrm{m}_1 : \tau_1]$ by the Amber rule one needs to show that $\mathrm{ref}_\circ(\alpha \to \tau_1) \subseteq \mathrm{ref}_\circ(\beta \to \tau_1)$, for any α and β such that $\alpha \subseteq \beta$. Clearly this does not hold. Even in a simpler setting without the reference types (*e.g.*, for the functional object calculus) the contravariance of the procedure type constructor in its first argument would cause subtyping to fail.

A combination of type recursion and an existential quantifier that uses the recursion variable as bound would allow us to enforce covariance for the positions of the recursion variable, and thus have subtyping in width:

$$[\mathrm{m}_d : \tau_d]_{d\in D} \stackrel{?}{=} \mu(\alpha).\exists\alpha'\subseteq\alpha.\{\mathrm{m}_d : \mathrm{ref}_\circ(\alpha' \to \tau_d)\}_{d\in D}$$

Intuitively α' can be viewed as the 'true' (*i.e.*, most precise) type of the object, while α is a more general type that can be given to it by subtyping. This is essentially the idea of the encodings of object types explored by Abadi *et al.* [2, 3].

For subtyping in depth with respect to the variance annotations we simply use the readable and writable reference types we defined in the previous section:

$$[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \stackrel{?}{=} \mu(\alpha).\exists\alpha'\subseteq\alpha.\{\mathrm{m}_d : \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d)\}_{d\in D}$$

Still, by keeping α' abstract, neither the typing rule for method invocation (Inv in Figure 5), nor the one for object cloning (Clone) is validated.

By explicitly enforcing in the definition of object types that the object value itself in fact belongs to this existentially quantified α', the assumptions become sufficiently strong to repair the invocation case. This is consistent with seeing α' as the 'true' type of the object. Semantically, we can express this using an intersection of types:

$$[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \stackrel{?}{=} \mu(\alpha).\exists\alpha'\subseteq\alpha.(\{\mathrm{m}_d : \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d)\}_{d\in D} \cap \alpha')$$

Forcing not only the current object value to be in α', but also all the 'sufficiently similar' values (maybe not even created yet), covers the case of cloning. The following definition formalizes this construction.

Definition 3.20 (Object types)**.** Let $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ be defined as the set of all triples $\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle$ such that $D \subseteq E$ and

$$\exists \alpha'.\ \alpha' \in \mathit{Type} \ \wedge\ \lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k \tag{Obj-1}$$

$$\wedge\, (\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d)) \tag{Obj-2}$$

$$\begin{aligned} \wedge\, (\forall j < k.\ \forall \Psi'.\ \forall \{\mathrm{m}_e{=}l'_e\}_{e\in E}\,. \qquad & \\ (k,\Psi) \sqsubseteq (j,\Psi')\ \wedge\ (\forall e \in E.\ \lfloor \Psi' \rfloor_j (l'_e) = \lfloor \Psi \rfloor_j (l_e)) & \\ \Rightarrow \langle j, \lfloor \Psi' \rfloor_j, \{\mathrm{m}_e{=}l'_e\}_{e\in E}\rangle \in \alpha') & \end{aligned} \tag{Obj-3}$$

The condition stating that $D \subseteq E$ ensures that all values in an object type provide *at least* the required methods listed by this type, but can also provide more. Clearly this is necessary for subtyping in width. Condition (Obj-1) postulates the existence of a more specific type α', the 'true' type of the object $\{\mathrm{m}_e{=}l_e\}_{e\in E}$ (up to approximation k), and the subsequent conditions are all stated in terms of α' rather than α. Condition (Obj-2) states the requirements for the methods in terms of the reference type constructors introduced in Section 3.4. Since the existentially quantified α' might equal α, one must take care that (Obj-2) does not introduce a circularity. However, due to the use of approximation in the definition of the reference type constructors, the condition only depends on $\lfloor \alpha' \rfloor_k$, rather than α'. This will ensure the well-foundedness of the construction.

As explained above, in order to invoke methods we must know that $\{\mathrm{m}_e{=}l_e\}_{e\in E}$ belongs to the more specific type α' for $j < k$ steps (which suffices since application consumes a step). In the particular case where Ψ' is Ψ and $\{\mathrm{m}_e{=}l'_e\}_{e\in E}$ is $\{\mathrm{m}_e{=}l_e\}_{e\in E}$ condition (Obj-3) states exactly this. We need the more general formulation in order to ensure that the clones of the considered object also belong to the same type α'. Therefore we enforce that no matter how an object value $\{\mathrm{m}_e{=}l'_e\}_{e\in E}$ is constructed, it belongs to type α' provided that it satisfies the same typing assumptions as $\{\mathrm{m}_e{=}l_e\}_{e\in E}$, with respect to a possibly extended heap typing Ψ'. Allowing for state extension is necessary since cloning itself allocates new locations not present in the original Ψ, and also because cloning can be performed after some intermediate computation steps that result in further allocations.

We show that this definition of object types actually makes sense, in that it defines a semantic type. This is not immediately obvious because of the recursion.

Proposition 3.21. *If $\tau_d \in \mathit{Type}$ for all $d \in D$, then we also have that $[m_d :_{\nu_d} \tau_d]_{d\in D} \in \mathit{Type}$.*

Proof sketch. We must show (1) that $[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ is well-defined, *i.e.*, that the recursive definition is well-founded, and (2) that it is closed under state extension.

To prove the well-definedness one can use general results about recursive types in step-indexed semantics [10], since the object type constructor is 'contractive'. Alternatively, from the observation that $\tau = \bigcup_k \lfloor \tau \rfloor_k$ for all types τ, it suffices to directly argue that Definition 3.20 defines $\lfloor [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \rfloor_k$ only in terms of $\lfloor [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \rfloor_j$ for $j < k$. The closure under state extension follows from the corresponding property of the types $\alpha' \to \tau_d$ (Proposition 3.14) and of the sets $\mathrm{ref}_{\nu_d}(\alpha' \to \tau_d)$ (Proposition 3.19), and from the transitivity of state extension. □

Let $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d \in D}$.

$$(\forall d \in D.\ \Sigma[x_d := \alpha] \models b_d : \tau_d) \implies \Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d \in D} : \alpha \qquad (\textsc{SemObj})$$

$$(\Sigma \models a : \alpha \ \wedge\ e \in D \ \wedge\ \nu_e \in \{+, \circ\}) \implies \Sigma \models a.\mathrm{m}_e : \tau_e \qquad (\textsc{SemInv})$$

$$(\Sigma \models a : \alpha \ \wedge\ e \in D \ \wedge\ \nu_e \in \{-, \circ\} \qquad (\textsc{SemUpd})$$

$$\wedge\ \Sigma[x := \alpha] \models b : \tau_e) \implies \Sigma \models a.\mathrm{m}_e := \varsigma(x)b : \alpha$$

$$\Sigma \models a : \alpha \implies \Sigma \models \mathrm{clone}\ a : \alpha \qquad (\textsc{SemClone})$$

$$(E \subseteq D \ \wedge\ (\forall e \in E.\ \nu_e \in \{+, \circ\} \Rightarrow \alpha_e \subseteq \beta_e) \qquad (\textsc{SemSubObj})$$

$$\wedge\ (\forall e \in E.\ \nu_d \in \{-, \circ\} \Rightarrow \beta_e \subseteq \alpha_e)) \implies [\mathrm{m}_d :_{\nu_d} \alpha_d]_{d \in D} \subseteq [\mathrm{m}_e :_{\nu_e} \beta_e]_{e \in E}$$

$$(\forall d \in D.\ \nu_d = \circ \ \vee\ \nu_d = \nu'_d) \implies [\mathrm{m}_d :_{\nu_d} \alpha_d]_{d \in D} \subseteq [\mathrm{m}_d :_{\nu'_d} \alpha_d]_{d \in D} \qquad (\textsc{SemSubObjVar})$$

Figure 8: Typing lemmas: object types

Figure 8 presents the semantic typing and subtyping lemmas for object types.

Lemma 3.22 (Object types)**.** *All the semantic typing lemmas shown in Figure 8 are valid implications.*

Proof sketch. The semantic typing lemmas are proved independently. We sketch this for (SemObj). A detailed proof, as well as the proofs of the other typing lemmas are given in the Appendix.

For $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d \in D}$ and assuming $\Sigma[x_d := \alpha] \models b_d : \tau_d$ for all $d \in D$, we must show that $\Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d \in D} : \alpha$. So let $k \geq 0$, σ and Ψ be such that $\sigma :_{k,\Psi} \Sigma$. By Definition 3.8 (Semantic typing judgement) we must prove that $\sigma([\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d \in D}) :_{k,\Psi} \alpha$, or equivalently (after suitable α-renaming), that $[\mathrm{m}_d{=}\varsigma(x_d)\sigma(b_d)]_{d \in D} :_{k,\Psi} \alpha$ holds. Now let h, h' and b' be such that $h :_k \Psi$ and

$$\langle h, [\mathrm{m}_d{=}\varsigma(x_d)\sigma(b_d)]_{d \in D} \rangle \to^j \langle h', b' \rangle$$

for some $j < k$, and assume that $\langle h', b' \rangle$ is irreducible. From the operational semantics it is clear that $j = 1$, $b' \equiv \{\mathrm{m}_d{=}l_d\}_{d \in D}$ and that, for some locations $l_d \notin dom(h)$,

$$h' = h\,[l_d := \lambda(x_d)\sigma(b_d)]_{d \in D}$$

Choosing $\Psi' = \left\lfloor \Psi\,[l_d := (\alpha \to \tau_d)]_{d \in D} \right\rfloor_{k-1}$ it is easily seen that $(k, \Psi) \sqsubseteq (k-1, \Psi')$. Furthermore, from the hypothesis by (SemLam) we have that $\Sigma \models \lambda(x_d)b_d : \alpha \to \tau_d$ for all $d \in D$. From this and the assumption that $h :_k \Psi$ it follows that $h' :_{k-1} \Psi'$.

By Definition 3.6 it remains to establish that $\langle k-1, \Psi', \{\mathrm{m}_d{=}l_d\}_{d \in D} \rangle \in \alpha$. This is achieved by proving the following more general claim by induction on j_0:

Claim 3.23. For all $j_0 \geq 0$, Ψ^* and $\{\mathrm{m}_d{=}l^*_d\}_{d \in D}$ we have that

$$(k-1, \Psi') \sqsubseteq (j_0, \Psi^*) \ \wedge\ (\forall d{\in}D.\, \lfloor \Psi^* \rfloor_{j_0}(l^*_d) = \left\lfloor \Psi' \right\rfloor_{j_0}(l_d)) \Rightarrow\ \langle j_0, \lfloor \Psi^* \rfloor_{j_0}, \{\mathrm{m}_d{=}l^*_d\}_{d \in D} \rangle \in \alpha$$

The key step is in choosing α' equal to $\lfloor \alpha \rfloor_{j_0}$, then verifying the three conditions of Definition 3.20 (Object types), where the inductive hypothesis is used for showing (Obj-3). □

Remark 3.24. In the above proof, establishing $\langle k{-}1, \Psi', \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle \in \alpha$ directly does not seem possible, and the generalization to Claim 3.23 arises naturally from a failed proof attempt: in order to prove the three conditions of Definition 3.20 a sensible choice for α' is α, and for E is D, after which (Obj-1) and (Obj-2) follow easily. But (Obj-3) requires us to show that $\langle j, \lfloor\Psi''\rfloor_j, \{\mathrm{m}_d{=}l'_d\}_{d\in D}\rangle \in \alpha$, for any $j < k$, any l'_d, and any extension Ψ'' of Ψ' with $\lfloor\Psi''\rfloor_j(l'_d) = \lfloor\Psi'\rfloor_j(l_d)$. This is just what Claim 3.23 states.

The fact that there is an inductive argument hidden in this proof does not come as a surprise: the induction on the step index j_0 resolves the recursion that is inherent to objects due to the self application semantics of method invocation.

3.6. **Bounded Quantified Types.** Impredicative quantified types were previously studied in a step-indexed setting by Ahmed *et al.* [6, 9] for a lambda-calculus with general references, and we follow their presentation. However, unlike in the work of Ahmed *et al.* our quantifiers have bounds, and we are also studying subtyping. It is important to note that the impredicative second-order types were the reason why a semantic stratification of types was needed in the presence of general references [6], as opposed to a syntactic one based on the nesting of reference types [8]. In the setting we consider in this paper we need the semantic stratification not only to explicitly accommodate quantified types, but also because our interpretation of object types uses existential types implicitly.

As in Appel and McAllester's work [10], a type constructor F (*i.e.*, a function from semantic types to semantic types) is non-expansive if in order to determine whether a term has type $F(\tau)$ with approximation k, it suffices to know the type τ only to approximation k. As we will later show (Lemma 3.33), all the type constructors we define in this paper are non-expansive.

Definition 3.25 (Non-expansiveness)**.** A type constructor $F : \mathit{Type} \to \mathit{Type}$ is *non-expansive* if for all types τ and for all $k \geq 0$ we have that $\lfloor F(\tau)\rfloor_k = \lfloor F(\lfloor\tau\rfloor_k)\rfloor_k$.

The definitions of second-order types require that $\forall$ and $\exists$ are only applied to non-expansive type constructors. The non-expansiveness condition ensures that in order to determine level k of a universal or existential type, quantification over the types in $\mathit{PreType}_k$ suffices. This helps avoid the circularity that is otherwise introduced by the *impredicative* quantification.

Definition 3.26 (Bounded universal types)**.** If $F : \mathit{Type} \to \mathit{Type}$ is non-expansive and $\alpha \in \mathit{Type}$, then we define $\forall_\alpha F$ by $\langle k, \Psi, \Lambda.\, a\rangle \in \forall_\alpha F$ if and only if

$$\forall j, \Psi'.\ \forall \tau.\ (k, \Psi) \sqsubseteq (j, \Psi')\ \wedge\ \tau \in \mathit{Type}\ \wedge\ \lfloor\tau\rfloor_j \subseteq \lfloor\alpha\rfloor_j \Rightarrow \forall i < j.\ a :_{i, \lfloor\Psi'\rfloor_i} F(\tau)$$

Definition 3.27 (Bounded existential types)**.** For all non-expansive $F : \mathit{Type} \to \mathit{Type}$ and $\alpha \in \mathit{Type}$, the set $\exists_\alpha F$ is defined by $\langle k, \Psi, \mathrm{pack}\ v\rangle \in \exists_\alpha F$ if and only if

$$\exists \tau.\tau \in \mathit{Type}\ \wedge\ \lfloor\tau\rfloor_k \subseteq \lfloor\alpha\rfloor_k\ \wedge\ \forall j < k.\ \langle j, \lfloor\Psi\rfloor_j, v\rangle \in F(\tau)$$

Proposition 3.28. *If $\alpha \in \mathit{Type}$ and $F : \mathit{Type} \to \mathit{Type}$ is non-expansive, then $\forall_\alpha F$ and $\exists_\alpha F$ are also types.* □

Proof sketch. The proofs are minor modifications of those given in [6], to additionally take the bounds into account. □

For all non-expansive $F, G : \mathit{Type} \to \mathit{Type}$,

$$(\forall \tau \in \mathit{Type}.\ \tau \subseteq \alpha \Rightarrow \Sigma \models a : F(\tau)) \implies \Sigma \models \Lambda.\, a : \forall_\alpha F \qquad \text{(SemTAbs)}$$

$$(\Sigma \models a : \forall_\alpha F \ \wedge\ \tau \in \mathit{Type} \ \wedge\ \tau \subseteq \alpha) \implies \Sigma \models a[] : F(\tau) \qquad \text{(SemTApp)}$$

$$(\exists \tau \in \mathit{Type}.\ \tau \subseteq \alpha \ \wedge\ \Sigma \models a : F(\tau)) \implies \Sigma \models \text{pack } a : \exists_\alpha F \qquad \text{(SemPack)}$$

$$(\Sigma \models a : \exists_\alpha F \ \wedge\ \forall \tau \in \mathit{Type}. \qquad \text{(SemOpen)}$$

$$\tau \subseteq \alpha \Rightarrow \Sigma[x := F(\tau)] \models b : \beta) \implies \Sigma \models \text{open } a \text{ as } x \text{ in } b : \beta$$

$$(\beta \subseteq \alpha \ \wedge\ \forall \tau \in \mathit{Type}.\ \tau \subseteq \beta \Rightarrow F(\tau) \subseteq G(\tau)) \implies \forall_\alpha F \subseteq \forall_\beta G \qquad \text{(SemSubUniv)}$$

$$(\alpha \subseteq \beta \ \wedge\ \forall \tau \in \mathit{Type}.\ \tau \subseteq \alpha \Rightarrow F(\tau) \subseteq G(\tau)) \implies \exists_\alpha F \subseteq \exists_\beta G \qquad \text{(SemSubExist)}$$

Figure 9: Typing lemmas: bounded quantified types

Lemma 3.29 (Bounded quantified types)**.** *All the semantic typing lemmas shown in Figure 9 are valid implications.*

Proof sketch. The first four implications are proved as in [6]; the additional precondition $\tau \subseteq \alpha$ in (SemTApp) and (SemPack) serves to establish the requirements for the bounds. The two subtyping lemmas (SemSubUniv) and (SemSubExist) are easily proved by just unfolding the definitions. □

3.7. **Recursive Types.** In contrast to most previous work on step-indexed models, we consider iso-recursive rather than equi-recursive types, so folds and unfolds are explicit in our syntax and consume computation steps. Iso-recursive types have been previously considered by Ahmed for a step-indexed relational model of the lambda calculus [7]. Iso-recursion is simpler, and sufficient for our purpose. As a consequence, we require type constructors to be only non-expansive, as opposed to the stronger 'contractiveness' requirement [10].

Definition 3.30 (Recursive types)**.** Let $F : \mathit{Type} \to \mathit{Type}$ be a non-expansive function. We define the set μF by

$$\langle k, \Psi, \text{fold } v\rangle \in \mu F \ \Leftrightarrow\ \forall j < k.\ \langle j, \Psi', v\rangle \in F(\mu F)$$

Proposition 3.31. *For all non-expansive $F : \mathit{Type} \to \mathit{Type}$, $\mu F \in \mathit{Type}$ is well-defined.*

Proof sketch. The well-definedness follows from the observation that $\lfloor \mu F \rfloor_k$ is defined only in terms of $\lfloor F(\mu F) \rfloor_j$ for $j < k$, which by the non-expansiveness of F means that $\lfloor \mu F \rfloor_k$ relies only on $\lfloor \mu F \rfloor_j$. The closure under state extension is established by an induction, proving that for each $k \geq 0$, $\lfloor \mu F \rfloor_k \in \mathit{Type}$. □

Figure 10 presents the semantic typing lemmas for recursive types. As a consequence, we have the expected fixed point property $\models a : F(\mu F) \ \Leftrightarrow \models \text{fold } a : \mu F$.

Lemma 3.32 (Recursive types)**.** *All the semantic typing lemmas shown in Figure 10 are valid implications.*

Proof sketch. The validity of (SemFold) and (SemUnfold) are easy consequences of Definition 3.30. For (SemSubRec), one shows by induction on k that $\lfloor \mu F \rfloor_k \subseteq \lfloor \mu G \rfloor_k$, using the precondition of the rule and the non-expansiveness of F and G. □

For all non-expansive $F, G : \mathit{Type} \to \mathit{Type}$,

$$\Sigma \models a : \mu F \implies \Sigma \models \mathrm{unfold}\, a : F(\mu F) \qquad (\textsc{SemUnfold})$$
$$\Sigma \models a : F(\mu F) \implies \Sigma \models \mathrm{fold}\, a : \mu F \qquad (\textsc{SemFold})$$
$$(\forall \alpha, \beta.\ \alpha \subseteq \beta \Rightarrow F(\alpha) \subseteq G(\beta)) \implies \mu F \subseteq \mu G \qquad (\textsc{SemSubRec})$$

Figure 10: Typing lemmas: recursive types

$$[\![X]\!]_\eta = \eta(X) \qquad [\![[\mathrm{m}_d :_{\nu_d} A_d]_{d\in D}]\!]_\eta = \left[\mathrm{m}_d :_{\nu_d} [\![A_d]\!]_\eta\right]_{d\in D}$$
$$[\![\mathit{Bot}]\!]_\eta = \bot \qquad [\![\mu(X)A]\!]_\eta = \mu(\lambda\alpha{\in}\mathit{Type}.\ [\![A]\!]_{\eta[X:=\alpha]})$$
$$[\![\mathit{Top}]\!]_\eta = \top \qquad [\![\forall(X{\leqslant}A)B]\!]_\eta = \forall_{[\![A]\!]_\eta}(\lambda\alpha{\in}\mathit{Type}.\ [\![B]\!]_{\eta[X:=\alpha]})$$
$$[\![A \to B]\!]_\eta = [\![A]\!]_\eta \to [\![B]\!]_\eta \qquad [\![\exists(X{\leqslant}A)B]\!]_\eta = \exists_{[\![A]\!]_\eta}(\lambda\alpha{\in}\mathit{Type}.\ [\![B]\!]_{\eta[X:=\alpha]})$$

Figure 11: Interpretation of types

Lemma 3.33 (Non-expansiveness)**.** *All the considered type constructors are non-expansive.*

Proof sketch. It is easily seen that the definition of $\lfloor \alpha \to \beta \rfloor_k$ uses only $\lfloor \alpha \rfloor_j$ and $\lfloor \beta \rfloor_j$ for $j < k$, and therefore that $\lfloor \alpha \to \beta \rfloor_k = \lfloor \lfloor \alpha \rfloor_k \to \lfloor \beta \rfloor_k \rfloor_k$. A similar statement holds for the k-th approximation of quantified types $\forall_\alpha F$ and $\exists_\alpha F$, since their definition only depends on $\lfloor \alpha \rfloor_j$ and $\lfloor F \rfloor_j$ for $j < k$. In the case of object and recursive types, the properties $\left\lfloor [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \right\rfloor_k = \left\lfloor [\mathrm{m}_d :_{\nu_d} \lfloor \tau_d \rfloor_k]_{d\in D} \right\rfloor_k$ and $\lfloor \mu F \rfloor_k = \lfloor \mu \lfloor F \rfloor_k \rfloor_k$ can be established by induction on k, using the non-expansiveness of F in the latter case. □

4. Semantic Soundness

In order to prove that well-typed terms are safe to evaluate we relate the syntactic types to their semantic counterparts, and then use the fact that the semantic typing judgement enforces safety by construction (Theorem 3.11). This approach is standard in denotational semantics. In fact, none of the main statements or proofs in this section mentions step-indices explicitly.

Definition 4.1 (Interpretation of types and typing contexts)**.** Let η be a total function from type variables to semantic types.

(1) The interpretation $[\![A]\!]_\eta$ of a type A is given by the structurally recursive meaning function defined in Figure 11.
(2) The interpretation of a well-formed typing context Γ with respect to η is given by the function that maps x to $[\![A]\!]_\eta$, for every $x{:}A \in \Gamma$.

Note that in Figure 11 the type constructors used on the left-hand sides of the equations are simply syntax, while those on the right hand-sides refer to the corresponding semantic constructions, as defined in the previous section.

Recall that non-expansiveness is a necessary precondition for some of the semantic typing lemmas. In particular, the well-definedness of $[\![A]\!]_\eta$ depends on non-expansiveness,

due to the use of μ, $\forall_{(\cdot)}$ and $\exists_{(\cdot)}$ in Figure 11. So we begin by showing that the interpretation of types is a non-expansive map.

Lemma 4.2 (Non-expansiveness)**.** $[\![A]\!]_\eta$ *is non-expansive in* η*.*

Proof sketch. We show that $\left\lfloor [\![A]\!]_\eta \right\rfloor_k = \left\lfloor [\![A]\!]_{\lfloor \eta \rfloor_k} \right\rfloor_k$ holds by induction on the structure of A, relying on Lemma 3.33 for the non-expansiveness of the semantic type constructions. □

Definition 4.3 ($\eta \models \Gamma$)**.** Let Γ be a well-formed typing context. We say that η *satisfies* Γ, written as $\eta \models \Gamma$, if $\eta(X) \subseteq [\![A]\!]_\eta$ holds for all $X \leqslant A$ appearing in Γ.

We show the soundness of the subtyping relation.

Lemma 4.4 (Soundness of subtyping)**.** *If* $\Gamma \vdash A \leqslant B$ *and* $\eta \models \Gamma$ *then* $[\![A]\!]_\eta \subseteq [\![B]\!]_\eta$*.*

Proof sketch. By induction on the derivation of $\Gamma \vdash A \leqslant B$ and case analysis on the last applied rule. Each case is immediately reduced to one of the subtyping lemmas from Section 3. □

Finally, we prove the semantic soundness of the syntactic type system with respect to the model.

Theorem 4.5 (Semantic soundness)**.** *Whenever* $\Gamma \vdash a : A$ *and* $\eta \models \Gamma$ *it follows that* $[\![\Gamma]\!]_\eta \models a : [\![A]\!]_\eta$*.*

Proof sketch. By induction on the derivation of $\Gamma \vdash a : A$ and case analysis on the last rule applied. Each case is easily reduced to one of the semantic typing lemmas from Section 3, using a standard type substitution lemma for derivations ending with an application of (Fold), (Unfold), (TApp), or (Pack). □

By Theorems 4.5 (Semantic soundness) and 3.11 (Safety), we have a proof of safety for the type system from Section 2.3.

Corollary 4.6 (Type safety)**.** *Well-typed terms are safe to evaluate.* □

5. Self Types

Self types have been proposed by Abadi and Cardelli [2] as a means to reconcile recursive object types with 'proper' subtyping. Self types are interesting because they allow us to type methods that return the possibly modified host object, or a clone of it. For instance, a type of list nodes, with a filter method that produces the sublist of all elements satisfying a given predicate, is

$$\mathit{List}_A = \mathit{Obj}(X)[\mathrm{hd} :_\circ A,\ \mathrm{tl} :_\circ X + \mathit{Unit},\ \mathrm{filter} :_+ (A \to \mathit{Bool}) \to X, \ldots]$$

Note that the similar recursive type

$$\mu(X)[\mathrm{hd} :_\circ A, \mathrm{tl} :_\circ X + \mathit{Unit}, \mathrm{filter} :_+ (A \to \mathit{Bool}) \to X, \ldots]$$

does not satisfy the usual subtyping for object types because of the invariance of the hd and tl fields.

Let $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D}$ and $\beta = [\mathrm{m}_e :_{\nu_e} G_e]_{e\in E}$ with $E \subseteq D$.

$$(\forall d \in D.\ \Sigma[x_d := \alpha] \models b_d : F_d(\alpha)) \implies \Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha \quad \text{(SemObj-Self)}$$

$$(\Sigma \models a : \alpha \ \wedge\ e \in D \ \wedge\ \nu_e \in \{+, \circ\}) \implies \Sigma \models a.\mathrm{m}_e : F_e(\alpha) \quad \text{(SemInv-Self)}$$

$$(\Sigma \models a : \alpha \ \wedge\ e \in D \ \wedge\ \nu_e \in \{-, \circ\} \quad \text{(SemUpd-Self)}$$
$$\wedge\ \forall \xi \subseteq \alpha.\ \Sigma[x := \xi] \models b : F_e(\xi)) \implies \Sigma \models a.\mathrm{m}_e := \varsigma(x)b : \alpha$$

$$\Sigma \models a : \alpha \implies \Sigma \models \mathrm{clone}\ a : \alpha \quad \text{(SemClone-Self)}$$

$$(\forall e \in E.\ (\nu_e \in \{+, \circ\} \Rightarrow \forall \xi \subseteq \alpha.\ F_e(\xi) \subseteq G_e(\xi)) \quad \text{(SemSubObj-Self)}$$
$$\wedge\ (\nu_e \in \{-, \circ\} \Rightarrow \forall \xi \subseteq \alpha.\ G_e(\xi) \subseteq F_e(\xi))) \implies \alpha \subseteq \beta$$

$$(\forall d \in D.\ \nu_d = \circ\ \vee\ \nu_d = \nu'_d) \implies [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D} \subseteq [\mathrm{m}_d :_{\nu'_d} F_d]_{d\in D} \quad \text{(SemSubObjVar-Self)}$$

Figure 12: Typing lemmas: self types

5.1. **Semantics of Self Types.** Abadi and Cardelli [2, Ch. 15] show how self types can be understood in terms of recursive and existentially quantified object types via an encoding. More precisely, the type $Obj(X)[\mathrm{m}_d :_{\nu_d} B_d]_{d\in D}$ where X may occur positively in B_d, stands for the recursive type $\mu(Y)\exists(X{\leqslant}Y)[\mathrm{m}_d :_{\nu_d} B_d]_{d\in D}$. The bounded existential quantifier introduced by this encoding gives rise to the desired subtyping in width and depth, despite the type recursion.

Since our type system features recursive and bounded existential types, self types could be accommodated via this encoding. However a treatment of self types can be achieved even more directly, without relying on the encoding. In fact, almost everything is in place already: recall that the semantics of object types (Definition 3.20) employs recursion and an existential quantification to refer to the 'true' type of an object. Condition Obj-2 in Definition 3.20 can be changed to take advantage of this type:

Definition 5.1 (Self types)**.** Assume $F_d : \mathit{Type} \to \mathit{Type}$ are monotonic and non-expansive type constructors, for all $d \in D$. Then let $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D}$ be defined as the set of all triples $\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle$ such that $D \subseteq E$ and

$$\exists \alpha'.\ \alpha' \in \mathit{Type} \ \wedge\ \lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k \quad \text{(Obj-1)}$$

$$\wedge\ (\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to F_d(\alpha'))) \quad \text{(Obj-2-self)}$$

$$\wedge\ (\forall j < k.\ \forall \Psi'.\ \forall \left\{\mathrm{m}_e{=}l'_e\right\}_{e\in E}. \quad \text{(Obj-3)}$$
$$(k, \Psi) \sqsubseteq (j, \Psi')\ \wedge\ (\forall e \in E.\ \left\lfloor \Psi' \right\rfloor_j (l'_e) = \lfloor \Psi \rfloor_j (l_e))$$
$$\Rightarrow \langle j, \left\lfloor \Psi' \right\rfloor_j, \left\{\mathrm{m}_e{=}l'_e\right\}_{e\in E}\rangle \in \alpha')$$

As in Section 3.5 one shows that Definition 5.1 uniquely determines a type. In this proof, the non-expansiveness of the type functions F_d is necessary in order to ensure that $\left\lfloor [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D} \right\rfloor_k$ is defined in terms of $\left\lfloor [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D} \right\rfloor_j$ for $j < k$ only. Moreover, the proofs of the typing lemmas for object types (see Section A.2 in the Appendix) carry over with minor modifications, to show that the semantic typing lemmas in Figure 12 hold. Most

cases are obtained by replacing the result type τ_d by $F_d(\alpha')$ throughout the proof, for α' the existentially quantified type from condition (Obj-1) of Definition 5.1. The proof of (SemInv-Self) uses the monotonicity of F_e, to conclude that the result of the invocation has type $F_e(\alpha)$ from the fact that it has type $F_e(\alpha')$, as given by condition (Obj-2-Self). In the proofs of (SemUpd-Self) and (SemSubObj-Self), the universally quantified ξ from the respective assumptions is instantiated by α'. Since (Obj-1) only gives that $\lfloor\alpha'\rfloor_k \subseteq \lfloor\alpha\rfloor_k$ but not necessarily $\alpha' \subseteq \alpha$, these three proofs also use the non-expansiveness of F_d in an essential way. Finally, given the non-expansiveness of each F_d, an induction shows that $\big\lfloor[\mathrm{m}_d :_{\nu_d} F_d]_{d\in D}\big\rfloor_k = \big\lfloor[\mathrm{m}_d :_{\nu_d} \lfloor F_d\rfloor_k]_{d\in D}\big\rfloor_k$ for all k. In other words, $[\mathrm{m}_d :_{\nu_d} F_d]_{d\in D}$, viewed as a type constructor, is non-expansive and Lemma 3.33 still holds.

The interpretation of syntactic type expressions given in Figure 11 extends straightforwardly to self types using the new type constructor:

$$\big[\!\big[\mathit{Obj}(X)[\mathrm{m}_d :_{\nu_d} A_d]_{d\in D}\big]\!\big]_\eta = \Big[\mathrm{m}_d :_{\nu_d} \lambda(\alpha{\in}\mathit{Type})\,[\![A_d]\!]_{\eta[X:=\alpha]}\Big]_{d\in D}$$

With this interpretation and the semantic typing lemmas from Figure 12, the soundness theorem from Section 4 should extend to a syntactic type system for objects with self types similar to the one derived by Abadi and Cardelli [2, Ch. 15] for their encoding (but also including variance annotations and a typing rule for cloning).

5.2. **Limitations.** Note that with the exception of SemUpd-Self, all the semantic typing lemmas for self types are stronger than their counterparts from Figure 8. This is already enough to typecheck many examples involving self types [2, Ch. 15].

However, as for the encoding of Abadi and Cardelli, when updating methods one usually does not have full information about the precise self type α' of the host object, which may be a proper subtype of α. Therefore the statement (SemUpd-Self) about method update in Figure 12 includes a quantification over all subtypes ξ of the known type α of the object a, to ensure that the updated method also works correctly for the precise type. As a consequence the new method body b must be sufficiently parametric in the type of its self parameter x, which can be overly restrictive. Abadi and Cardelli [2, Ch. 17] demonstrate this limitation with an example of objects that provide a backup and a retrieve method:

$$\mathit{Bk} = \mathit{Obj}(X)[\text{retrieve} :_\circ X,\ \text{backup} :_\circ X, \ldots]$$

A sensible definition of the backup method updates the retrieve method so that a subsequent invocation of retrieve yields a clone of the current object x:

$$\mathit{backup}(x) \;=\; \text{let } z = \text{clone } x \text{ in } x.\text{retrieve} \coloneqq \varsigma(y)z$$

Here, the 'let $z = a$ in b' stands for the usual syntactic sugar $(\lambda(z)b)\,a$. Let $\beta = [\![\mathit{Bk}]\!]_\eta$ denote the interpretation of the syntactic type *Bk*. While the backup method has the correct operational behaviour, to typecheck the method update to x in its body using (SemUpd-Self) we would need the statement $\Sigma\,[x{:=}\beta, z{:=}\beta, y{:=}\xi] \models z : \xi$. But this statement does not hold for an arbitrary subtype $\xi \subseteq \beta$. Therefore the semantic typing lemmas stated above are not strong enough to prove that $\Sigma\,[x{:=}\beta, z{:=}\beta] \models x.\text{retrieve} \coloneqq \varsigma(y)z : \beta$, and thus that $\Sigma\,[x{:=}\beta] \models \mathit{backup}(x) : \beta$ holds for the method body. This prevents us from typing an object that contains this method (*e.g.*, $[\text{backup} = \varsigma(x)\mathit{backup}(x), \ldots]$) to type β using the semantic typing lemmas.

Abadi and Cardelli [2, Ch. 17] address this lack of expressiveness by modifying the calculus in two respects. First, they introduce a new syntax for method update, $a.\mathrm{m} \coloneqq (y, z = c)\varsigma(x)b$. Operationally this new construct behaves just like

$$\text{let } y = a \text{ in let } z = c \text{ in } y.\mathrm{m} \coloneqq \varsigma(x)b \tag{5.1}$$

but its typing rule is more powerful than the one induced by this encoding. When typing c and the method body b, y can be assumed to have the precise type of the object:

$$(\textsc{Upd-Self})\ \frac{\begin{array}{c} A \equiv Obj(Y)[\mathrm{m}_d :_{\nu_d} A_d]_{d\in D} \qquad \Gamma \vdash a : A \qquad e \in D \qquad \nu_e \in \{-, \circ\} \\ \Gamma, X{\leqslant}A, y{:}X \vdash c : C \qquad \Gamma, X{\leqslant}A, y{:}X, z{:}C, x{:}X \vdash b : A_e \end{array}}{\Gamma \vdash a.\mathrm{m}_e \coloneqq (y, z = c)\varsigma(x)b : A}$$

Second, in order to propagate this information, typing rules with 'structural' assumptions are introduced. For instance, the inference rule for object cloning takes the form

$$(\textsc{Clone-Str})\ \frac{A{\leqslant}Obj(Y)[\mathrm{m}_d :_{\nu_d} A_d]_{d\in D} \qquad \Gamma \vdash a : A}{\Gamma \vdash \text{clone } a : A}$$

thus applying also in the case where A is a type variable. In this modified system, the body of the backup method can be rewritten as

$$\mathit{backup}_{\mathit{mod}}(x) \;=\; x.\text{retrieve} \coloneqq (y, z = \text{clone } y)\varsigma(x)z \tag{5.2}$$

and the judgement $\Gamma, x{:}\mathit{Bk} \vdash \mathit{backup}_{\mathit{mod}}(x) : \mathit{Bk}$ is derivable.

Even in the purely syntactic setting, the *ad hoc* character of the syntax extension is not entirely satisfactory, but for the step-indexed semantics of types both modifications are in fact problematic. First, although it seems reasonable to expect that all the semantic typing lemmas from Section 3 continue to hold, a change of the calculus and its operational semantics would require us to recheck the proofs about object types in detail. Fortunately, the syntax extension does not seem necessary from the semantic typing point of view; we can already prove the semantic soundness of rule (Upd-Self) with respect to the encoding of the new method update construct from (5.1):
If $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D}$, $e \in D$, and $\nu_e \in \{-, \circ\}$ then

$$\Sigma \models a : \alpha \;\wedge\; \forall \xi \subseteq \alpha.\, \Sigma[y \coloneqq \xi] \models c : \gamma \;\wedge\; \forall \xi \subseteq \alpha.\, \Sigma\,[y \coloneqq \xi, z \coloneqq \gamma, x \coloneqq \xi] \models b : F_e(\xi)$$
$$\Longrightarrow \quad \Sigma \models \text{let } y = a \text{ in let } z = c \text{ in } y.\mathrm{m} \coloneqq \varsigma(x)b : \alpha$$

However, by itself this rule does not help in typing the body of the backup method, and the introduction of rules with structural assumptions presents a more severe difficulty. Soundness of these rules relies on the fact that every subtype of an object type is another object type. In other words, in the (Clone-Str) rule the type A is assumed to range only over object types. Such structural assumptions are usually not valid in semantic models, and they are certainly not justified with respect to our semantically defined subtype relation, which is just set inclusion.

5.3. **Self Types with Structural Assumptions.** To sum up the previous subsection, the problem is that the semantic typing lemmas from Figure 12 are too weak to type certain examples such as the body of the backup method, but the usual way to strengthen these rules in a syntactic setting is not semantically sound in our model. Still, $\Sigma[x \coloneqq \beta] \models \mathit{backup}(x) : \beta$ is a valid typing judgement about the method body. This can be seen by taking a closer

Let $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$.

$$(\forall d{\in}D.\ \forall \xi{\in}\mathit{Type}.\ \xi \lhd \alpha \Rightarrow \Sigma[x_d := \xi] \models b_d : F_d(\xi)) \implies \Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d \in D} : \alpha \quad \text{(SemObj-Str)}$$

$$(\alpha' \lhd \alpha \ \wedge\ \Sigma \models a : \alpha' \ \wedge\ e \in D \ \wedge\ \nu_e \in \{+, \circ\}) \implies \Sigma \models a.\mathrm{m}_e : F_e(\alpha') \quad \text{(SemInv-Str)}$$

$$(\alpha' \lhd \alpha \ \wedge\ \Sigma \models a : \alpha' \ \wedge\ e \in D \ \wedge\ \nu_e \in \{-, \circ\} \quad \text{(SemUpd-Str)}$$

$$\wedge\ \Sigma[x := \alpha'] \models b : F_e(\alpha')) \implies \Sigma \models a.\mathrm{m}_e \coloneqq \varsigma(x)b : \alpha'$$

$$\alpha' \lhd \alpha \ \wedge\ \Sigma \models a : \alpha' \implies \Sigma \models \text{clone } a : \alpha' \quad \text{(SemClone-Str)}$$

$$\Sigma \models a : \alpha \ \wedge\ (\forall \xi{\in}\mathit{Type}.\ \xi \lhd \alpha \Rightarrow \Sigma[x := \xi] \models b : \beta) \implies \Sigma \models \text{let } x = a \text{ in } b : \beta \quad \text{(SemLet-Str)}$$

Figure 13: Typing lemmas with structural assumptions: self types

look at the semantic definition of the self type $\beta = [\![Bk]\!]_\eta$: essentially, if for a suitable substitution $\sigma :_{k,\Psi} \Sigma[x := \beta]$ the substitution instance

$$\sigma(\mathit{backup}(x)) = \text{let } z = \text{clone } \sigma(x) \text{ in } \sigma(x).\text{retrieve} \coloneqq \varsigma(y)z$$

becomes irreducible in less than k steps, then $\sigma(x)$ must be an object value $v = \{\mathrm{m}_d{=}l_d\}_{d \in D}$ such that $\langle k, \Psi, v\rangle \in \beta$. Property (Obj-1) of β asserts the existence of a type α' such that $\lfloor \alpha' \rfloor_k \subseteq \lfloor \beta \rfloor_k$, and property (Obj-3) entails that z becomes bound to a value v' of this type α'. Thus by (Obj-2-self) the eventual update of the retrieve field of v is valid, since the new method $\lambda(y)v'$ has the expected type $\alpha' \to \alpha'$ to sufficient approximation.

Similar 'manual' reasoning seems possible in other cases, but a more principled approach will let us use typing lemmas that are strong enough and avoid explicit reasoning about the operational semantics and step indices. To facilitate this, we develop a semantic counterpart to the structural assumptions that appear in the syntactic type system of Abadi and Cardelli [2, Ch. 17]. More precisely, we introduce a relation $\alpha' \lhd \alpha$ between semantic types that strengthens the subtype relation: intuitively α' is the precise, recursive type of some collection of object values from the object type α. The type α acts as an interface that lists the permitted operations on these object values.

Definition 5.2 (Self type exposure)**.** For $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$ and $\alpha' \in \mathit{Type}$ the relation $\alpha' \lhd \alpha$ holds if and only if $\alpha' \subseteq \alpha$ and for all $E \supseteq D$ and $\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e \in E}\rangle \in \alpha'$,

$$(\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to F_d(\alpha'))) \quad \text{(Obj-2-self)}$$

$$\wedge\quad (\forall j < k.\ \forall \Psi'.\ \forall \left\{\mathrm{m}_e{=}l'_e\right\}_{e \in E}. \quad \text{(Obj-3)}$$

$$(k, \Psi) \sqsubseteq (j, \Psi') \ \wedge\ (\forall e \in E.\ \left\lfloor \Psi' \right\rfloor_j (l'_e) = \lfloor \Psi \rfloor_j (l_e))$$

$$\Rightarrow \langle j, \left\lfloor \Psi' \right\rfloor_j, \left\{\mathrm{m}_e{=}l'_e\right\}_{e \in E}\rangle \in \alpha')$$

Notice that $\alpha' \lhd \alpha$ essentially states that α' is a type that can take the place of the existentially quantified 'self type' in an object type (see Definition 5.1). It is immediate from this definition that $\alpha' \lhd \alpha$ implies $\alpha' \subseteq \alpha$. Note however that $\lhd$ is not reflexive: in general, α' is not an object type (*e.g.*, α' could be empty). Intuitively, the object type α is obtained as a union of such α'.

Figure 13 lists new typing lemmas for self types that exploit the relation $\lhd$. Compared to (SemInv-Self) and (SemClone-Self) from Figure 12, the typing lemmas (SemInv-Str) and (SemClone-Str) use the additional assumptions $\alpha' \lhd \alpha$ and $\Sigma \models a : \alpha'$ to establish a more precise typing for the result. Similarly, while (SemUpd-Self) universally quantifies over all $\xi \subseteq \alpha$ in its premise, (SemUpd-Str) limits this to those $\xi \in \mathit{Type}$ for which $\xi \lhd \alpha$ holds. Finally, (SemLet-Str) lets us use an object a within b with the more precise type ξ where $\xi \lhd \alpha$, and similarly (SemObj-Str) lets us type the method bodies under the more informative assumption that $\xi \lhd \alpha$. The latter two are the key lemmas to introduce an assumption $\alpha' \lhd \alpha$ in proofs using these semantic typing lemmas.

As an illustration, consider the example of the backup method again. In order to construct objects with the backup method, we will establish that

$$\forall \xi \in \mathit{Type}.\ \xi \lhd \beta \Rightarrow \Sigma[x := \xi] \models \mathit{backup}(x) : \xi \tag{5.3}$$

holds, where $\beta = [\![Bk]\!]_\eta$ and $\mathit{backup}(x)$ abbreviates 'let $z =$ clone x in x.retrieve $:= \varsigma(y)z$' as before. From this, by (SemObj-Str) it will follow that

$$\Sigma \models [\text{backup} = \varsigma(x)\mathit{backup}(x), \text{retrieve} = \ldots] : \beta$$

If we desugar the let construct in $\mathit{backup}(x)$ and apply lemmas (SemApp) and (SemLam), we notice that in order to show (5.3) it suffices to prove that $\Sigma[x{:=}\xi] \models$ clone $x : \xi$ and $\Sigma[x{:=}\xi, z{:=}\xi] \models x$.retrieve $:= \varsigma(y)z : \xi$. Using $\xi \lhd \beta$ and $\Sigma[x{:=}\xi] \models x : \xi$, the validity of the former judgement is immediate by (SemClone-Str). Similarly, the latter follows by (SemUpd-Str) from the fact that $\xi \lhd \beta$ and since the retrieve method is listed with variance annotation '$\circ$' in β.

Lemma 5.3 (Self types: lemmas with structural assumptions)**.** *All the semantic typing lemmas shown in Figure 13 are valid implications.*

Proof sketch. The proofs of (SemInv-Str), (SemClone-Str), and (SemUpd-Str) are straightforward adaptations of those for (SemInv), (SemClone), and (SemUpd). As an example, we give the proof of (SemUpd-Str) as Lemma A.10 in the Appendix. More interestingly, (SemLet-Str) relies on the following property of object types α:

$$\langle k, \Psi, v\rangle \in \alpha \quad \Longrightarrow \quad \exists \alpha' \in \mathit{Type}.\ \alpha' \lhd \alpha \ \wedge\ \langle k-1, \lfloor \Psi \rfloor_{k-1}, v\rangle \in \alpha'$$

In the proof of (SemLet-Str), this α' is used to instantiate the universally quantified ξ in the premise $\xi \lhd \alpha \Rightarrow \Sigma[x := \xi] \models b : \beta$. The full proof is given as Lemma A.12 in the Appendix.

The proof of (SemObj-Str) is similar to the one of (SemObj) (Lemma A.4 in the Appendix), except that we use the heap typing extension $\Psi' = \big\lfloor \Psi\,[l_d := (\beta \to F_d(\beta))]_{d\in D} \big\rfloor_{k-1}$, where β is a recursive record type satisfying conditions (Obj-2-self) and (Obj-3), but not validating any subtyping property. In verifying that the extended heap is well-typed with respect to this Ψ', one uses that $\beta \lhd \alpha$, in order to instantiate the assumptions on the method bodies and obtain $\Sigma[x_d := \beta] \models b_d : F_d(\beta)$. Finally, to show that the generated object value has type α, Claim 3.23 is strengthened to show that the object value is in fact in β, which is a subtype of α since $\beta \lhd \alpha$ (see Proposition A.14 and Lemma A.15 in the Appendix for the full proof). □

Remark 5.4. The implication $\Sigma \models a : \alpha \implies \exists \alpha' \in \mathit{Type}.\ \alpha' \lhd \alpha \ \wedge\ \Sigma \models a : \alpha'$ for $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$ may appear reasonable (and would entail both (SemLet-Str) and (SemObj-Str)), but we do not believe that it holds. The problem is that, while the premise $\Sigma \models a : \alpha$ guarantees for each $k \geq 0$ the existence of a type α'_k that satisfies the requirement $\alpha'_k \lhd \alpha$, it is in general not possible to construct a type α' 'in the limit' from this sequence. For the same reason, the implication $\Sigma \models \mathrm{pack}\ a : \exists_\alpha F \implies \exists \alpha' \subseteq \alpha.\, \Sigma \models a : F(\alpha')$ is not valid. The typing lemma (SemLet-Str) avoids this problem since α' is only needed up to a fixed approximation, and so the choice of α'_k for sufficiently large k suffices (*cf.* proof of Lemma A.12 in the Appendix). On the other hand, the (SemObj-Str) lemma avoids the problem by instantiating ξ with a particular type β, for which $\beta \lhd \alpha$ is already known.

In this section we showed that our semantics of object types naturally extends to self types, while avoiding any change to the syntax and operational semantics of the calculus. We proved a first set of typing lemmas that are natural and apply to many examples (Figure 12). These lemmas are however not sufficient to typecheck self-returning methods. To achieve this, we developed a second set of typing lemmas that involve the object's precise type, through the relation $\alpha' \lhd \alpha$ (Figure 13). Note that these latter lemmas do not fully subsume the former ones, since the $\lhd$ relation is not reflexive. We leave open the problem of relating the lemmas in Figure 13 to a syntactic type system.

6. Generalizing Reference and Object Types

The semantics described in this paper generalizes the reference types from [6, 9] to readable and writable reference types. This can be generalized even further. We can have a reference type constructor that takes *two* types as arguments: one that represents the most general type that can be used when writing to the reference, and another for the most specific type that can be read from it [38]. This can be easily expressed using our readable and writable reference types together with intersection types:

$$\mathrm{ref}(\tau^w, \tau^r) = \mathrm{ref}_- \tau^w \cap \mathrm{ref}_+ \tau^r$$

After unfolding the definitions, this yields

$$\mathrm{ref}(\tau^w, \tau^r) = \{\langle k, \Psi, l\rangle \mid \lfloor \tau^w \rfloor_k \subseteq \lfloor \Psi(l) \rfloor_k \subseteq \lfloor \tau^r \rfloor_k\}.$$

As one would expect, this generalized reference type constructor is contravariant in the first argument and covariant in the second one:

$$\beta^w \subseteq \alpha^w \quad \wedge \quad \alpha^r \subseteq \beta^r \qquad \implies \qquad \mathrm{ref}(\alpha^w, \alpha^r) \subseteq \mathrm{ref}(\beta^w, \beta^r) \qquad \text{(SemSubRef-Gen)}$$

Note that, if one takes these generalized reference types as primitive, then the three reference types from Section 3.4 are obtained as special cases:

$$\mathrm{ref}_\circ \tau = \mathrm{ref}(\tau, \tau), \qquad \mathrm{ref}_+ \tau = \mathrm{ref}(\bot, \tau), \qquad \mathrm{ref}_- \tau = \mathrm{ref}(\tau, \top),$$

and the subtyping properties from Figure 7 are still valid.

Figures 14 and 15 give a graphical representation of the different reference type constructors. In both figures the horizontal axis represents the type at which a reference can be read, while the vertical one gives the type at which it can be written. Notice that because of the different variance the read axis goes from $\bot$ to $\top$ while the write axis from $\top$ to $\bot$.

Figure 14 represents the usual, as well as the readable, and the writable reference types as points on the three edges of a triangle. Notice that the usual references can be read and written at the same type. Without additional information, the readable references

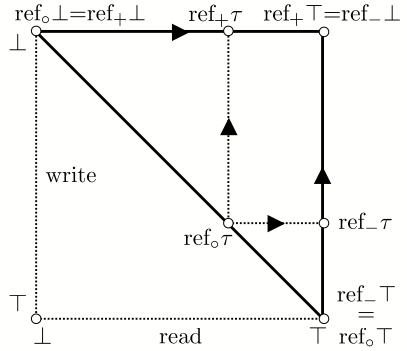

Figure 14: Readable/writable reference types

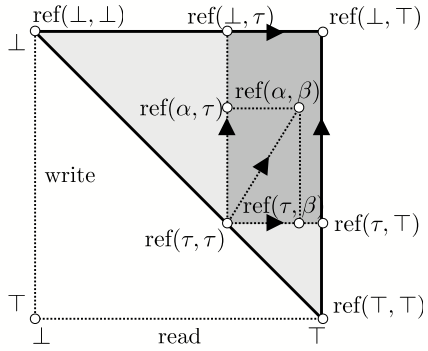

Figure 15: Generalized reference types

Let $\alpha = [\mathrm{m}_d : (\tau_d^w, \tau_d^r)\,]_{d\in D}$ and $\alpha' = [\mathrm{m}_d : (\tau_d, \tau_d)\,]_{d\in D}$.

$$(\forall d \in D.\ \Sigma[x_d := \alpha'] \models b_d : \tau_d) \implies \Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha' \quad \text{(SemObj-Gen)}$$

$$(\Sigma \models a : \alpha \ \wedge\ e \in D) \implies \Sigma \models a.\mathrm{m}_e : \tau_e^r \quad \text{(SemInv-Gen)}$$

$$(\Sigma \models a : \alpha \ \wedge\ e \in D \ \wedge\ \Sigma[x := \alpha] \models b : \tau_e^w) \implies \Sigma \models a.\mathrm{m}_e := \varsigma(x)b : \alpha \quad \text{(SemUpd-Gen)}$$

$$\Sigma \models a : \alpha \implies \Sigma \models \mathrm{clone}\ a : \alpha \quad \text{(SemClone-Gen)}$$

$$(E \subseteq D \ \wedge\ (\forall e \in E.\ \beta_e^w \subseteq \alpha_e^w \ \wedge\ \alpha_e^r \subseteq \beta_e^r)) \implies [\mathrm{m}_d : (\alpha_d^w, \alpha_d^r)\,]_{d\in D} \subseteq [\mathrm{m}_e : (\beta_e^w, \beta_e^r)\,]_{e\in E} \quad \text{(SemSubObj-Gen)}$$

Figure 16: Typing lemmas: generalized object types

can only be written safely at type $\bot$, and the writable ones can only be read at type $\top$. Subtyping is represented by arrows: covariant on the edge of the readable reference types and contravariant on the writable reference types' edge. An invariant reference type can only be subtyped either to a readable or to a writable reference type.

Figure 15 illustrates that our generalization of reference types is indeed very natural. When generalizing, we take not only the points on the edges of the triangle, but also the points inside it to be reference types. Furthermore, instead of having three different kinds of reference types, we only have one. Subtyping is also more natural: the set of all supertypes of a reference type cover the area of a rectangle which goes from the point corresponding to this reference type to the 'top' reference type $\mathrm{ref}(\bot, \top)$. For instance, the dark gray rectangle in Figure 15 contains all supertypes of $\mathrm{ref}(\tau, \tau)$.

Applying this idea in the context of the imperative object calculus leads not only to more expressive subtyping but also to simplifications, since the variance annotations are no longer needed. The extended object type $[\mathrm{m}_d : (\tau_d^w, \tau_d^r)\,]_{d\in D}$ has two types for each method m_d: τ_d^w is the most general type that can be used to update the given method, and τ_d^r is the most specific type that can be expected as a result when invoking the method. When defining the semantics of these generalized object types, the only difference with respect to Definition 3.20 (Object types) is that condition (Obj-2) is changed to use an extended reference type:

$$\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}(\alpha' \to \tau_d^w, \alpha' \to \tau_d^r). \quad \text{(Obj-2-Gen)}$$

Let $A = [\mathrm{m}_d : (A^w_d, A^r_d)\,]_{d\in D}$ and $A' = [\mathrm{m}_d : (A_d, A_d)\,]_{d\in D}$.

$$\text{(Obj-Gen)}\ \frac{\forall d \in D.\ \Gamma, x_d : A' \vdash b_d : A_d}{\Gamma \vdash \left[\mathrm{m}_d = \varsigma(x_d{:}A')b_d\right]_{d\in D} : A'} \qquad \text{(Clone-Gen)}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{clone}\ a : A}$$

$$\text{(Inv-Gen)}\ \frac{\Gamma \vdash a : A \quad e \in D}{\Gamma \vdash a.\mathrm{m}_e : A^r_e} \qquad \text{(Upd-Gen)}\ \frac{\Gamma \vdash a : A \quad e \in D \quad \Gamma, x : A \vdash b : A^w_e}{\Gamma \vdash a.\mathrm{m}_e \coloneqq \varsigma(x{:}A)b : A}$$

$$\text{(SubObj-Gen)}\ \frac{E \subseteq D \quad \forall e \in E.\ \Gamma \vdash B^w_e \leqslant A^w_e \quad \forall e \in E.\ \Gamma \vdash A^r_e \leqslant B^r_e}{\Gamma \vdash [\mathrm{m}_d : (A^w_d, A^r_d)\,]_{d\in D} \leqslant [\mathrm{m}_e : (B^w_e, B^r_e)\,]_{e\in E}}$$

Figure 17: The typing rules for generalized object types

Figure 16 presents the semantic typing lemmas that are validated by this definition of object types, while Figure 17 gives the corresponding syntactic typing rules. Note that the complex and seemingly ad-hoc rules for subtyping object types given in Figure 4 or in [2] are replaced by only one rule (SubObj-Gen).

Lemma 6.1 (Generalized object types)**.** *All the semantic typing lemmas shown in Figure 16 are valid implications.*

Proof sketch. The proof of the subtyping lemma (SemSubObj-Gen) follows easily from the lemma for subtyping generalized reference types (SemSubRef-Gen above), and is therefore significantly simpler than when variance annotations are involved (see Lemma A.16 in the Appendix). For all the other semantic typing lemmas the proofs are basically unchanged (see Section A.2 in the Appendix). □

Note that the generalization of object types presented in this section is orthogonal to the extension to self types from the previous section. The generalized object types lead to a type system that is both simpler and more expressive than the usual type systems for objects [2]. Our generalized object types directly correspond to the *split types* of Bugliesi and Pericás-Geertsen [19], who have shown that these types are strictly more expressive than object types with variance annotations [19, Example 4.3].

7. Comparison to Related Work

7.1. **Domain-theoretic Models.** Abadi and Cardelli give a semantic model for the functional object calculus in [1, 2]. Their type system is comparable to the one we consider here. Types are interpreted as certain partial equivalence relations over an untyped domain-theoretic model of the calculus. No indication is given on how to adapt this to the imperative execution model.

Based on earlier work by Kamin and Reddy [30], Reus *et al.* [41, 42, 44] construct domain-theoretic models for the imperative object calculus, with the goal of proving soundness for the logic of Abadi and Leino [4]. The higher-order store exhibited by the object calculus requires defining the semantic domains by mixed-variant recursive equations. The dynamic allocation is then addressed by interpreting specifications of the logic as Kripke relations, indexed by store specifications, which are similar to the heap typings used here.

Building on work by Levy [31], an 'intrinsically typed' model of the imperative object calculus is presented in the second author's PhD thesis [44], by solving the domain equations in a suitable category of functors. However, only first-order types are considered.

Compared to these domain-theoretic models, the step-indexed model we present not only soundly interprets a richer type language, but is also easier to work with. The way it is based on the operational semantics eliminates the need for explicit continuity conditions, and the admissibility conditions are replaced by the closure under state extension, which is usually very easy to check. All that is needed for the definition of iso-recursive and second-order types are non-expansiveness and the stratification invariant. What is missing from our model is a semantic notion of equality that approximates program equivalence. For reasoning about program equivalence in an ML-like language, Ahmed *et al.* [5] have recently developed a *relational* step-indexed model, and it could be interesting to adapt their work to an object-oriented setting.

Recently proposed models for polymorphism and general references [15, 16, 17] suggest that an adequate semantics for imperative objects with expressive typing could in principle be developed also in a domain-theoretic setting. A detailed comparison between step-indexed semantics and domain-theoretic models would be useful, to make the similarities and differences between the two approaches more precise.

It is interesting to see how the object construction rule (Obj) is proved correct in each of the models described above. In the domain-theoretic self-application models [41, 42], it directly corresponds to a recursive predicate whose well-definedness (*i.e.*, existence and uniqueness) must be established. This proof exploits properties of the underlying, recursively defined domain, and imposes some further restrictions on the semantic types: besides admissibility, types appearing in the defining equation of a recursive predicate need to satisfy an analogue of the contractiveness property [36]. In the typed functor category model [44], object construction is interpreted using a recursively defined function, and correspondingly (Obj) is proved by fixed point induction. In the step-indexed case, the essence of the proof is a more elementary induction on the step index, with a suitably generalized induction hypothesis (see Claim 3.23 in the proof sketch of Lemma 3.22 on page 18, or the full proof in the Appendix).

7.2. **Interpretations of Object Types.** Our main contribution in this paper is the novel interpretation of object types in the step-indexed model. The step-index-induced stratification permits the construction of mixed-variance recursive as well as impredicative, second-order types. Both are key ingredients in our interpretation of object types. The use of recursive and existentially quantified types is in line with the type-theoretic work on object encodings, which however has mainly focused on object calculi with a functional execution model [18].

Closest to our work is the encoding of imperative objects into an imperative variant of system $F_{\leqslant\mu}$ with updatable records, proposed by Abadi *et al.* [3]. There, objects are interpreted as records containing references to the procedures that represent the methods. As in our case, these records have a recursive and existentially quantified record type. The difference is that two additional record fields are included in order to achieve invocation and cloning, and uninitialized fields are used to construct this recursive record. Subtyping in depth is considered in [3] only for the encoding of the functional object calculus. However, if one added to the target language the readable and writable reference types we use in this

paper, the encoding of the imperative object calculus would extend to subtyping in depth as well.

In the typing rules for self types, the structural assumptions about the subtype relation play an important role [2]. In Section 5 we developed a semantic counterpart to such typing rules with structural assumptions, in order to deal with the polymorphic update of self-returning methods. This is, however, tailored specifically to object types. Hofmann and Pierce [26] investigate the metatheory of subtyping with structural assumptions in general, and give elementary presentations of two encodings of functional objects in a variant of System $F_{\leq}$ with type destructors. It may be interesting to see if a step-indexed model of this variant of System $F_{\leq}$ can be found.

7.3. **Step-indexed Models.** Step-indexed semantic models were introduced by Appel *et al.* in the context of foundational proof-carrying code. Their goal was to construct more elementary and modular proofs of type soundness that can be easily checked automatically. They were primarily interested in low-level languages, however they also applied their technique to a pure λ-calculus with recursive types [10]. Later Ahmed *et al.* successfully extended it to general references and impredicative polymorphism [6, 9]. The step-indexed semantic model we present extends the one by Ahmed *et al.* with object types and subtyping. In order to achieve this, we refine the reference types from [6] to readable and writable reference types.

Subtyping in a step-indexed semantic model was previously considered by Swadi who studied Typed Machine Language [45]. Our setup is however much different. In particular, the subtle issues concerning the subtyping of object types are original to our work.

The previous work on step-indexing focuses on 'semantic type systems', *i.e.*, the semantic typing lemmas can directly be used for type-checking programs [9, 10, 12]. However, when one considers more complex type systems with subtyping, recursive types or polymorphism, the semantic typing lemmas no longer directly correspond to the usual syntactic rules. These discrepancies can be fixed, but usually at the cost of more complex models, like the one developed by Swadi to track type variables [12, 45]. In Swadi's model an additional 'semantic kind system' is used to track the contractiveness and non-expansiveness of types with free type variables. We avoid having a more complex model (*e.g.*, one that tracks type variables) by considering iso-recursive rather than equi-recursive types. An equi-recursive type is well-defined if its argument is contractive, and some of the type constructors are not contractive in general (*e.g.*, the identity as well as the equi-recursive type constructor itself). On the other hand, an iso-recursive type is well-defined under the weaker assumption that the argument is non-expansive, and all our type constructors are indeed non-expansive (see Lemma 3.33). It is then relatively straightforward to use the semantic typing lemmas in order to prove the soundness of the standard, syntactic type system we consider (see Theorem 4.5).

7.4. **Type Safety Proofs.** Abadi and Cardelli use subject reduction to prove the safety of several type systems very similar to the one considered in this paper [2]. Those purely syntactic proofs are very different from the 'semantic' type safety proof we present (for detailed discussions about the differences see [10, 46]). Since type safety is built into the model, our safety proof neither relies on a preservation property, nor can preservation be concluded from it.

Constructing a step-indexed semantics is more challenging than proving progress and preservation. However, for our particular semantics we could reuse the model by Ahmed *et al.* and extend it to suit our needs, even though the calculus we are considering is quite different. So one would expect that once enough general models are constructed (*e.g.*, [6, 10, 11]), it will become easier to build new models just by mixing and matching. Assuming the existence of an adequate step-indexed model, the effort needed to prove the semantic typing lemmas using 'pencil-and-paper' is somewhat comparable to the one required for a subject reduction proof. Since each of the semantic typing lemmas is proved in isolation, the resulting type soundness proof is more modular; the extensions we consider in Sections 5 and 6 illustrate this aspect rather well. According to Appel's original motivation, the advantages of step-indexing should become even more apparent when formalizing the proofs in a proof assistant [10].

7.5. **Generalized Reference and Object Types.** The readable and the writable reference types we define in Section 3.4 and use for modeling object types in Section 3.5 are similar to the reference types in the Forsythe programming language [43] and to the channel types of [22, 35]. The generalization to a reference type constructor taking two arguments described in Section 6 is quite natural, and also appeared in Pottier's thesis [38], where it facilitated type inference by allowing meets and joins to distribute over reference types. This idea has recently been applied by Craciun *et al.* for inferring variant parametric types in Java [23].

The generalized object types we introduce in Section 6 directly correspond to the *split types* of Bugliesi and Pericás-Geertsen [19]. Split types are also motivated by type inference, since they guarantee the existence of more precise upper and lower bounds. In particular, Bugliesi and Pericás-Geertsen show that split types are strictly more expressive than first-order object types with variance annotations [19, Example 4.3]. They establish the soundness of a type system with split types by subject reduction, with respect to a functional semantics of the object calculus.

7.6. **Functional Object Calculus.** Our initial experiments on the current topic were done in the context of the functional object calculus [27]. Even though in the functional setting the semantic model is much simpler, both models satisfy the same semantic typing lemmas. Even more, the syntactic type system we considered for the functional calculus is exactly the same as the one in this paper, so all the results in Section 4 directly apply to the functional object calculus: well-typed terms do not get stuck, no matter whether they are evaluated in a functional or an imperative way. It would not be possible to directly prove such a result using subject reduction, since for subject reduction the syntactic typing judgment for the imperative calculus would also depend on a heap typing, and thus be different from the judgment for the functional calculus. However, since we are not using subject reduction, we do not need to type-check partially evaluated terms that contain heap locations.

8. Conclusion

We have presented a step-indexed semantics for Abadi and Cardelli's imperative object calculus, and used it to prove the safety of a type system with object types, recursive and second-order types, as well as subtyping. We showed how this semantics can be extended to self types and typing lemmas with structural assumptions; and generalized in a way that

eliminates the need for variance annotations and at the same time simplifies the subtyping rules for objects.

The step-indexing technique is however not limited to type safety proofs, and has already been employed for more general reasoning about programs. Based on previous work by Appel and McAllester [10], Ahmed built a step-indexed partial equivalence relation model for the lambda calculus with recursive and impredicative quantified types, and showed that her relational interpretation of types is sound for proving contextual equivalences [7]. Recently, this was extended significantly to reason about program equivalence in the presence of general references [5]. Benton also used step-indexing as a technical device, together with a notion of orthogonality relating expressions to contexts, to show the soundness of a compositional program logic for a simple stack-based abstract machine [13]. He also employed step-indexing in a Floyd-Hoare-style framework based on relational parametricity for the specification and verification of machine code programs [14].

We hope that our work paves the way for similarly compelling, semantic investigations of program logics for the imperative object calculus: using a step-indexed model it should be possible to prove the soundness of more expressive program logics for this calculus.

Acknowledgements

We express our gratitude to the anonymous reviewers for their detailed and constructive comments on the preliminary versions of this article. We also thank Andreas Rossberg for pointing us to the work of John C. Reynolds on Forsythe. Cătălin Hriţcu is supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science.

References

[1] Martín Abadi and Luca Cardelli. A semantics of object types. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science, LICS'94*, pages 332–341. IEEE Computer Society Press, 1994.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Proceedings 23rd Symposium on Principles of Programming Languages, POPL'96*, pages 396–409. ACM Press, 1996.

[4] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice. Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*, Lecture Notes in Computer Science, pages 11–41. Springer, 2004.

[5] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings 36th Symposium on Principles of Programming Languages, POPL'09*, pages 340–353, 2009.

[6] Amal J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[7] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Proceedings 15th European Symposium on Programming, ESOP'06*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[8] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings 17th Annual IEEE Symposium Logic in Computer Science, LICS'02*, pages 75–86. IEEE Computer Society Press, 2002.

[9] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. Princeton University, January 2003.

[10] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[11] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Proceedings 34th Symposium on Principles of Programming Languages, POPL'07*, pages 109–122, 2007.

[12] Andrew W. Appel, Christopher Richards, and Kedar Swadi. A kind system for typed machine language. Technical report, Princeton University, September 2002.

[13] Nick Benton. A typed, compositional logic for a stack-based abstract machine. In Zoltán Ésik, editor, *Proceedings Asian Symposium on Programming Languages and Systems, APLAS'05*, volume 3780 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2005.

[14] Nick Benton. Abstracting allocation: the new new thing. In Zoltán Ésik, editor, *Proceedings Computer Science Logic, CSL'06*, volume 4207 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

[15] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. In *Proceedings Foundations of Software Science and Computation Structures, FOSSACS'09*, volume 5504 of *Lecture Notes in Computer Science*, pages 456–470. Springer, 2009.

[16] Nina Bohr. *Advances in Reasoning Principles for Contextual Equivalence and Termination*. PhD thesis, IT University of Copenhagen, 2007.

[17] Nina Bohr and Lars Birkedal. Relational reasoning for recursive types and references. In Naoki Kobayashi, editor, *Proceedings Asian Symposium on Programming Languages, APLAS'06*, volume 4279 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2006.

[18] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

[19] Michele Bugliesi and Santiago M. Pericás-Geertsen. Type inference for variant object types. *Information and Computation*, 177(1):2–27, 2002.

[20] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 21–47. Springer, 1985.

[21] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[22] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the π-calculus. *Theoretical Computer Science*, 398(1-3):217–242, 2008. Essays in honour of Mario Coppo, Mariangiola Dezani-Ciancaglini and Simona Ronchi della Rocca.

[23] Florin Craciun, Wei-Ngan Chin, Guanhua He, and Shengchao Qin. An interval-based inference of variant parametric types. In Giuseppe Castagna, editor, *Proceedings 18th European Symposium on Programming, ESOP '09*, volume 5502 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2009.

[24] Cormac Flanagan, Stephen Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *Workshop on Foundations and Developments of Object-Oriented Languages, FOOL/WOOD'06*, 2006.

[25] Andrew D. Gordon, Paul D. Hankin, and Søren B. Lassen. Compilation and equivalence of imperative objects. In S. Ramesh and G. Sivakumar, editors, *Proceedings 17th Conference on Foundations of Software Technology and Theoretical Computer Science, FST+TCS'97*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer, 1997.

[26] Martin Hofmann and Benjamin C. Pierce. Type destructors. *Information and Computation*, 172(1):29–62, 2002.

[27] Cătălin Hriţcu. A step-indexed semantic model of types for the functional object calculus. Master's thesis, Programming Systems Lab, Saarland University, May 2007.

[28] Cătălin Hriţcu and Jan Schwinghammer. A step-indexed semantics of imperative objects. Extended version, Programming Systems Lab, Saarland University, February 2008.

[29] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theoretical Computer Science*, 338(1-3):17–63, 2005.

[30] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.
[31] Paul Blain Levy. Possible world semantics for general storage in call-by-value. In Julian Bradfield, editor, *Proceedings Computer Science Logic, CSL'02*, volume 2471 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2002.
[32] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):99–124, 1991.
[33] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, 1990.
[34] Frank J. Oles. Type algebras, functor categories, and block structure. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambrige University Press, 1985.
[35] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.
[36] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
[37] Andrew M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher-Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.
[38] François Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report 3483, INRIA, September 1998.
[39] Uday S. Reddy and Hongseok Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.
[40] Bernhard Reus. Modular semantics and logics of classes. In Matthias Baatz and Johann A. Makowsky, editors, *Proceedings Computer Science Logic, CSL'03*, volume 2803 of *Lecture Notes in Computer Science*, pages 456–469. Springer, 2003.
[41] Bernhard Reus and Jan Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 16(2):313–358, April 2006.
[42] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. *Theoretical Computer Science*, 316:191–213, 2004.
[43] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173-233, Birkhäuser, 1997.
[44] Jan Schwinghammer. *Reasoning about Denotations of Recursive Objects*. PhD thesis, Department of Informatics, University of Sussex, 2006.
[45] Kedar N. Swadi. *Typed Machine Language*. PhD thesis, Princeton University, July 2003.
[46] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendix A.

A.1. **Auxiliary Propositions.**

Proposition A.1 (Preorder)**.** *The state extension relation, $\sqsubseteq$, is reflexive and transitive.* □

Proposition A.2 (Information-forgetting extension)**.** *If $j \leq k$ then $(k, \Psi) \sqsubseteq (j, \lfloor \Psi \rfloor_j)$.* □

Proposition A.3 (Relation between $\langle k, \Psi, v\rangle \in \tau$ and $v :_{k,\Psi} \tau$)**.** *Let v be a closed value.*
(1) *If $\langle k, \Psi, v\rangle \in \tau$ then $v :_{k,\Psi} \tau$.*
(2) *If $v :_{k,\Psi} \tau$, $k > 0$, and there exists some h such that $h :_k \Psi$, then $\langle k, \Psi, v\rangle \in \tau$.* □

A.2. **Typing Lemmas for Object Types.**

Lemma A.4 (SemObj: Object construction)**.** *For all object types* $\alpha = [m_d :_{\nu_d} \tau_d]_{d\in D}$*, if for all* $d \in D$ *we have* $\Sigma[x_d := \alpha] \models b_d : \tau_d$*, then* $\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha$*.*

Proof. Let $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ and assume that $\forall d \in D.\ \Sigma[x_d := \alpha] \models b_d : \tau_d$. We must show that $\Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha$. Thus, let $k \geq 0$, σ be a value environment and Ψ be a heap typing such that $\sigma :_{k,\Psi} \Sigma$. By the definition of the semantic typing judgement (Definition 3.8) we need to show that $\sigma([\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D}) :_{k,\Psi} \alpha$. Equivalently (after suitable α-renaming), we show that

$$[\mathrm{m}_d{=}\varsigma(x_d)\sigma(b_d)]_{d\in D} :_{k,\Psi} \alpha$$

Suppose $j < k, h, h'$ and b' are such that the following three conditions are fulfilled:

$$h :_k \Psi \quad \wedge \quad \langle h, [\mathrm{m}_d{=}\varsigma(x_d)\sigma(b_d)]_{d\in D}\rangle \to^j \langle h', b'\rangle \quad \wedge \quad \langle h', b'\rangle \not\to \tag{A.1}$$

By the operational semantics Red-Obj is the only rule that applies, which means that necessarily $j = 1$ and for some distinct $l_d \notin dom(h)$ we have $b' = \{\mathrm{m}_d{=}l_d\}_{d\in D}$ and

$$h' = h\,[l_d := \lambda(x_d)\sigma(b_d)]_{d\in D} \tag{A.2}$$

We choose

$$\Psi' = \lfloor \Psi\,[l_d := (\alpha \to \tau_d)]_{d\in D} \rfloor_{k-1} \tag{A.3}$$

and show that

$$(k, \Psi) \sqsubseteq (k-1, \Psi') \quad \wedge \quad h' :_{k-1} \Psi' \quad \wedge \quad \langle k-1, \Psi', b'\rangle \in \alpha \tag{A.4}$$

That the first conjunct of (A.4) holds is immediate from the construction of Ψ' (A.3).

In order to show the second conjunct, by Definition 3.5 (Well-typed heap) we first need to show that $dom(\Psi') \subseteq dom(h')$. From the first conjunct of (A.1) and Definition 3.5 it is clear that $dom(\Psi) \subseteq dom(h)$. Thus from the shape of h' (A.2) and the definition of Ψ' (A.3) we obtain the required inclusion.

Next, let $i < k-1$ and $l \in dom(\Psi')$. To establish $h' :_{k-1} \Psi'$ in (A.4) we now need to show that $\langle i, \lfloor \Psi' \rfloor_i, h'(l)\rangle \in \Psi'(l)$. We distinguish two cases:

- Case $l = l_d$ for some $d \in D$. From (A.2) and (A.3) respectively we get that

$$h'(l) = \lambda(x_d)\sigma(b_d) \quad \wedge \quad \Psi'(l) = \lfloor \alpha \to \tau_d \rfloor_{k-1}$$

Thus we need to show that

$$\langle i, \lfloor \Psi' \rfloor_i, \lambda(x_d)\sigma(b_d)\rangle \in \lfloor \alpha \to \tau_d \rfloor_{k-1} \tag{A.5}$$

By SemLam in Figure 6 (Lemma 3.15) and the assumption $\Sigma[x_d := \alpha] \models b_d : \tau_d$ we already know that $\Sigma \models \lambda(x_d)b_d : \alpha \to \tau_d$ for all $d \in D$. From this and $\sigma :_{k,\Psi} \Sigma$ by Definition 3.8 (Semantic typing judgement) we obtain

$$\forall d \in D.\ \lambda(x_d)\sigma(b_d) \ :_{k,\Psi} \ \alpha \to \tau_d \tag{A.6}$$

Since $k > 1$ and from (A.1) $h :_k \Psi$, Proposition A.3 shows that (A.6) implies

$$\forall d \in D.\ \langle k, \Psi, \lambda(x_d)\sigma(b_d)\rangle \in \alpha \to \tau_d \tag{A.7}$$

By Proposition A.2 we get that $(k-1, \Psi') \sqsubseteq (i, \lfloor \Psi' \rfloor_i)$, which together with the first conjunct of (A.4) and the transitivity of $\sqsubseteq$ yields $(k, \Psi) \sqsubseteq (i, \lfloor \Psi' \rfloor_i)$. Since each $\alpha \to \tau_d$ is closed under state extension, the latter property and (A.7) imply the required (A.5).

- Case $l \in dom(\Psi)$. From (A.2) and (A.3) respectively we get that $h'(l) = h(l)$ and $\Psi'(l) = \lfloor \Psi(l) \rfloor_{k-1}$, so we actually need to show that $\langle i, \lfloor \Psi' \rfloor_i, h(l) \rangle \in \lfloor \Psi(l) \rfloor_{k-1}$. From $h :_k \Psi$ (A.1) by Definition 3.5 we get that $\langle k-1, \lfloor \Psi \rfloor_{k-1}, h(l) \rangle \in \Psi(l)$. Since $\Psi(l)$ is closed under state extension and $(k-1, \lfloor \Psi \rfloor_{k-1}) \sqsubseteq (i, \lfloor \Psi' \rfloor_i)$, we obtain $\langle i, \lfloor \Psi' \rfloor_i, h(l) \rangle \in \Psi(l)$.

Finally, we need to show the third conjunct of (A.4), *i.e.*, $\langle k-1, \Psi', \{\mathrm{m}_d{=}l_d\}_{d \in D} \rangle \in \alpha$. To this end, we prove the following more general claim:
Claim: For all $j_0 \geq 0$, for all Ψ_0 and for all $\{\mathrm{m}_d{=}l'_d\}_{d \in D}$

$$(k-1, \Psi') \sqsubseteq (j_0, \Psi_0) \ \wedge \ (\forall d \in D.\ \lfloor \Psi_0 \rfloor_{j_0}(l'_d) = \lfloor \Psi' \rfloor_{j_0}(l_d)) \\ \Rightarrow \ \langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, \{\mathrm{m}_d{=}l'_d\}_{d \in D} \rangle \in \alpha \quad \text{(A.8)}$$

From this and $\lfloor \Psi' \rfloor_{k-1} = \Psi'$, (A.4) follows by taking $j_0 = k-1$, $\Psi_0 = \Psi'$, and $l'_d = l_d$ for all $d \in D$, and by observing that $\sqsubseteq$ is reflexive (Proposition A.1).

The claim is proved by complete induction on j_0. So assume $j_0 \geq 0$ and Ψ_0 are such that

$$(k-1, \Psi') \sqsubseteq (j_0, \Psi_0) \quad \text{(A.9)}$$

Moreover, for all $d \in D$ let $l'_d \in dom(\Psi_0)$ such that

$$\forall d \in D.\ \lfloor \Psi_0 \rfloor_{j_0}(l'_d) = \lfloor \Psi' \rfloor_{j_0}(l_d) \quad \text{(A.10)}$$

We show that $\langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, \{\mathrm{m}_d{=}l'_d\}_{d \in D} \rangle \in \alpha$, by checking that all the conditions obtained by unfolding the definition of $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d \in D}$ hold. Choosing $\alpha' = \lfloor \alpha \rfloor_{j_0}$ yields (Obj-1):

$$\exists \alpha'.\alpha' \in \mathit{Type} \ \wedge \ \lfloor \alpha' \rfloor_{j_0} \subseteq \lfloor \alpha \rfloor_{j_0} \quad \text{(A.11)}$$

Next, by the construction of Ψ' in (A.3), together with (A.9), (A.10), and the non-expansiveness of procedure types, it follows that for all $d \in D$

$$\lfloor \Psi_0 \rfloor_{j_0}(l'_d) = \lfloor \Psi' \rfloor_{j_0}(l_d) = \lfloor \alpha \to \tau_d \rfloor_{j_0} = \left\lfloor \lfloor \alpha \rfloor_{j_0} \to \tau_d \right\rfloor_{j_0} = \lfloor \alpha' \to \tau_d \rfloor_{j_0} \quad \text{(A.12)}$$

By the definition of reference types (Definition 3.16) this implies that

$$\forall d \in D.\ \langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, l'_d \rangle \in \mathrm{ref}_\circ(\alpha' \to \tau_d) \quad \text{(A.13)}$$

By the lemma for subtyping variance annotations (SemSubVarRef in Figure 7) we then obtain property (Obj-2):

$$\forall d \in D.\ \langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, l'_d \rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d) \quad \text{(A.14)}$$

Finally, we must prove (Obj-3), *i.e.*, that for all $j < j_0$, Ψ_1 and $\{\mathrm{m}_d{=}l''_d\}_{d \in D}$

$$(j_0, \Psi_0) \sqsubseteq (j, \Psi_1) \ \wedge \ (\forall d \in D.\ \lfloor \Psi_1 \rfloor_j(l''_d) = \lfloor \Psi_0 \rfloor_j(l'_d)) \\ \Rightarrow \ \langle j, \lfloor \Psi_1 \rfloor_j, \{\mathrm{m}_d{=}l''_d\}_{d \in D} \rangle \in \alpha \quad \text{(A.15)}$$

Note that this last condition holds trivially in the base case of the induction, when $j_0 = 0$. So assume $j < j_0$ and Ψ_1 and l''_d are such that $(j_0, \Psi_0) \sqsubseteq (j, \Psi_1)$ and $\lfloor \Psi_1 \rfloor_j(l''_d) = \lfloor \Psi_0 \rfloor_j(l'_d)$ for all $d \in D$. Now $j < j_0$ and assumption (A.10) yield that for all $d \in D$

$$\lfloor \Psi_1 \rfloor_j(l''_d) = \lfloor \Psi_0 \rfloor_j(l'_d) = \left\lfloor \lfloor \Psi_0 \rfloor_{j_0}(l'_d) \right\rfloor_j = \left\lfloor \lfloor \Psi' \rfloor_{j_0}(l_d) \right\rfloor_j = \lfloor \Psi' \rfloor_j(l_d)$$

Moreover, from $(k-1,\Psi') \sqsubseteq (j_0,\Psi_0)$ (A.9) and $(j_0,\Psi_0) \sqsubseteq (j,\Psi_1)$, by the transitivity of $\sqsubseteq$ we have that $(k-1,\Psi') \sqsubseteq (j,\Psi_1)$. Since $j < j_0$, the induction hypothesis of the claim gives

$$\langle j, \lfloor \Psi_1 \rfloor_j, \{\mathrm{m}_d{=}l''_d\}_{d\in D}\rangle \in \alpha$$

and we have established (A.15).

By Definition 3.20 applied to the object type $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ the properties $D \subseteq D$, (A.11), (A.14), and (A.15) establish that indeed $\langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, \{\mathrm{m}_d{=}l'_d\}_{d\in D}\rangle \in \alpha$. This finishes the inductive proof of claim (A.8), and the proof of the lemma. □

Lemma A.5 (SemInv: Method invocation)**.** *For all object types $\alpha = [m_d :_{\nu_d} \tau_d]_{d\in D}$ and for all $e \in D$, if $\Sigma \models a : \alpha$ and $\nu_e \in \{+,\circ\}$, then $\Sigma \models a.m_e : \tau_e$.*

Proof. Let $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$. We assume that $e \in D$, $\nu_e \in \{+,\circ\}$, and $\Sigma \models a : \alpha$ and show that $\Sigma \models a.\mathrm{m}_e : \tau_e$. To this end, let $k \geq 0$, σ and Ψ such that $\sigma :_{k,\Psi} \Sigma$. From $\Sigma \models a : \alpha$ by Definition 3.8 we get that

$$\sigma(a) :_{k,\Psi} \alpha \tag{A.16}$$

We need to show that $\sigma(a).\mathrm{m}_e :_{k,\Psi} \tau_e$. Thus, let $j < k$, and consider heaps h and h' and a term b' such that the following three conditions are fulfilled:

$$h :_k \Psi \quad \wedge \quad \langle h, \sigma(a).\mathrm{m}_e\rangle \to^j \langle h', b'\rangle \quad \wedge \quad \langle h', b'\rangle \not\to \tag{A.17}$$

From the second and third conjunct of (A.17) by the operational semantics we have that for some $i \leq j$, h^* and b^*

$$\langle h, \sigma(a)\rangle \to^i \langle h^*, b^*\rangle \not\to \quad \wedge \quad \langle h^*, b^*.\mathrm{m}\rangle \to^{j-i} \langle h', b'\rangle \tag{A.18}$$

From the first conjunct together with (A.16) and the first conjunct of (A.17), by Definition 3.6 it follows that there exists a heap typing Ψ^* such that

$$(k,\Psi) \sqsubseteq (k-i,\Psi^*) \quad \wedge \quad h^* :_{k-i} \Psi^* \quad \wedge \quad \langle k-i, \Psi^*, b^*\rangle \in \alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \tag{A.19}$$

By the definition of object types, the latter shows that there exists C and α' such that $b^* = \{\mathrm{m}_c{=}l_c\}_{c\in C}$, $D \subseteq C$ and (Obj-1) and (Obj-2) hold:

$$\alpha' \in \mathit{Type} \quad \wedge \quad \lfloor \alpha' \rfloor_{k-i} \subseteq \lfloor \alpha \rfloor_{k-i} \tag{A.20}$$

$$\forall d \in D.\ \langle k-i, \Psi^*, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d) \tag{A.21}$$

as well as (Obj-3): for all $j_0 < k-i$, all Ψ' and all $\{\mathrm{m}_c{=}l'_c\}_{c\in C}$,

$$((k-i,\Psi^*) \sqsubseteq (j_0,\Psi') \ \wedge\ \forall c \in C.\ \lfloor \Psi' \rfloor_{j_0}(l'_c) = \lfloor \Psi^* \rfloor_{j_0}(l_c)) \Rightarrow \langle j_0, \lfloor \Psi' \rfloor_{j_0}, \{\mathrm{m}_c{=}l'_c\}_{c\in C}\rangle \in \alpha' \tag{A.22}$$

From $e \in D$ and $\nu_e \in \{+,\circ\}$ using (A.21) we deduce that $\lfloor \Psi^* \rfloor_{k-i}(l_e) \subseteq \lfloor \alpha' \to \tau_e \rfloor_{k-i}$. So by expanding the definition of $h^* :_{k-i} \Psi^*$ from (A.19) for $k-i-1 < k-i$ we have

$$\langle k-i-1, \lfloor \Psi^* \rfloor_{k-i-1}, h^*(l_e)\rangle \in \lfloor \alpha' \to \tau_e \rfloor_{k-i} \tag{A.23}$$

By the definition of the procedure type $\alpha' \to \tau_e$ this means in particular that $h^*(l_e)$ must be an abstraction, *i.e.*, for some x and a', $h^*(l_e) = \lambda(x)a'$. Thus, since $\{\mathrm{m}_c{=}l_c\}_{c\in C} \in \mathit{CVal}$

and $e \in D \subseteq C$, by (A.18), RED-CTX, RED-INV, RED-BETA and the operational semantics, we obtain a reduction sequence of the form

$$\begin{aligned}\langle h, \sigma(a).\mathrm{m}_e\rangle &\to^{i} \langle h^*, \{\mathrm{m}_c{=}l_c\}_{c\in C}.\mathrm{m}_e\rangle \\ &\to \langle h^*, (\lambda(x)a')\ \{\mathrm{m}_c{=}l_c\}_{c\in C}\rangle \\ &\to \langle h^*, \{\!\{x \mapsto \{\mathrm{m}_c{=}l_c\}_{c\in C}\}\!\}(a')\rangle \\ &\to^{j-i-2} \langle h', b'\rangle\end{aligned} \tag{A.24}$$

Since $k-i-2 < k-i$ by Proposition A.2 we have that

$$(k-i, \Psi^*) \sqsubseteq (k-i-2, \lfloor\Psi^*\rfloor_{k-i-2}) \tag{A.25}$$

We can now use the property (OBJ-3) of the object type α: we instantiate (A.22) with $l'_c = l_c$, $j_0 = k-i-2$, and $\Psi' = \lfloor\Psi^*\rfloor_{k-i-2}$ to obtain

$$\langle k-i-2, \lfloor\Psi^*\rfloor_{k-i-2}, \{\mathrm{m}_c{=}l_c\}_{c\in C}\rangle \in \alpha' \tag{A.26}$$

From this using (A.23) and from the definition of procedure types, it follows that

$$\{\!\{x \mapsto \{\mathrm{m}_c{=}l_c\}_{c\in C}\}\!\}(a') :_{k-i-2, \lfloor\Psi^*\rfloor_{k-i-2}} \tau_e \tag{A.27}$$

On the other hand, the second conjunct of (A.19) implies

$$h^* :_{k-i-2} \lfloor\Psi^*\rfloor_{k-i-2} \tag{A.28}$$

by Definition 3.5, Proposition A.2 and the closure of types under state extension. Moreover, by (A.24), $\langle h^*, \{\!\{x \mapsto \{\mathrm{m}_c{=}l_c\}_{c\in C}\}\!\}(a')\rangle \to^{j-i-2} \langle h', b'\rangle$, which by (A.17) is irreducible. This, combined with (A.28) and (A.27), by Definition 3.6, means that there exists Ψ'' such that

$$(k-i-2, \lfloor\Psi^*\rfloor_{k-i-2}) \sqsubseteq (k-j, \Psi'') \quad\wedge\quad h' :_{k-j} \Psi'' \quad\wedge\quad \langle k-j, \Psi'', b'\rangle \in \tau_e \tag{A.29}$$

From the first conjunct above, the first conjunct in (A.19), and (A.25), using the transitivity of state extension we obtain

$$(k, \Psi) \sqsubseteq (k-j, \Psi'') \tag{A.30}$$

From (A.17), (A.30), and the second and third conjuncts of (A.29), by Definition 3.6 we can conclude that $\sigma(a).\mathrm{m}_e :_{k,\Psi} \tau_e$ holds. This is what we needed to show. □

Lemma A.6 (SEMUPD: Method update)**.** *For all object types $\alpha = [m_d :_{\nu_d} \tau_d]_{d\in D}$ and for all $e \in D$, if $\Sigma \models a : \alpha$ and $\Sigma[x := \alpha] \models b : \tau_e$ and $\nu_e \in \{-, \circ\}$, then $\Sigma \models a.m_e := \varsigma(x)b : \alpha$.*

Proof sketch. The proof is similar to that of Lemma A.5 (Method invocation). The existence of some Ψ^* such that

$$\langle h, \sigma(a).\mathrm{m}_e := \varsigma(x)\sigma(b)\rangle \to^{j} \langle h^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}.\mathrm{m}_e := \varsigma(x)\sigma(b)\rangle$$

with $(k, \Psi) \sqsubseteq (k-j, \Psi^*)$, $h^* :_{k-j} \Psi^*$ and $\langle k-j, \Psi^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha$ follows from the existence of a corresponding reduction sequence $\langle h, \sigma(a)\rangle \to^{j} \langle h^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle$. Since the only reduction from $\langle h^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}.\mathrm{m}_e := \varsigma(x)\sigma(b)\rangle$ is by (RED-UPD) and results in the configuration $\langle h', \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle$ where

$$h' \;=\; h^*\,[l_e := \lambda(x)\sigma(b)] \tag{A.31}$$

the proof of the lemma is essentially a matter of showing that $h' :_{k-j-1} \lfloor\Psi^*\rfloor_{k-j-1}$.

First, note that $dom(\Psi^*) \subseteq dom(h') = dom(h^*)$ holds, by $h^* :_{k-j} \Psi^*$. Next, let $i < k-j-1$, and let $l \in dom(\Psi^*)$. It remains to show that

$$\langle i, \lfloor \Psi^* \rfloor_i, h'(l) \rangle \in \lfloor \Psi^*(l) \rfloor_{k-j-1} \tag{A.32}$$

Note that by the definition of $\langle k-j, \Psi^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha$ (Definition 3.20) it follows that there exists $\alpha' \in \mathit{Type}$ such that $\lfloor \alpha' \rfloor_{k-j} \subseteq \lfloor \alpha \rfloor_{k-j}$, that $D \subseteq E$, and that

$$\forall d \in D.\ \langle k-j, \Psi^*, l_d \rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d) \tag{A.33}$$

We now prove (A.32) by a case distinction on the location l:

- Case $l = l_e$. From (A.31) we have that $h'(l_e) = \lambda(x)\sigma(b)$. Since $e \in D \subseteq E$ and $\nu_e \in \{-, \circ\}$ by assumption, (A.33) yields $\lfloor \alpha' \to \tau_e \rfloor_{k-j} \subseteq \lfloor \Psi^* \rfloor_{k-j}(l_e)$. Since $\lfloor \alpha' \rfloor_{k-j} \subseteq \lfloor \alpha \rfloor_{k-j}$, the subtyping lemma (SemSubProc) and the non-expansiveness of procedure types yield $\lfloor \alpha \to \tau_e \rfloor_{k-j} \subseteq \lfloor \Psi^* \rfloor_{k-j}(l_e)$. The monotonicity of semantic approximation therefore entails
$$\lfloor \alpha \to \tau_e \rfloor_{k-j-1} \subseteq \lfloor \Psi^* \rfloor_{k-j-1}(l_e) \tag{A.34}$$
Additionally, the assumption $\Sigma[x := \alpha] \models b : \tau_e$ gives $\lambda(x)\sigma(b) :_{k-j,\Psi^*} \alpha \to \tau_e$. Since $0 \le i < k-j$ and $h^* :_{k-j} \Psi^*$, Proposition A.3 yields $\langle k-j, \Psi^*, \lambda(x)\sigma(b)\rangle \in \alpha \to \tau_e$. By the closure under state extension, this implies $\langle i, \lfloor \Psi^* \rfloor_i, \lambda(x)\sigma(b)\rangle \in \alpha \to \tau_e$, from which (A.32) follows by (A.34).
- Case $l \neq l_e$. This case is easier since the value in the heap does not change for this location, *i.e.*, $h'(l) = h^*(l)$, so the result follows from the closure under state extension of $\Psi^*(l)$. □

Lemma A.7 (SemClone: Object cloning)**.** *For all object types* $\alpha = [m_d :_{\nu_d} \tau_d]_{d\in D}$, *if* $\Sigma \models a : \alpha$ *then* $\Sigma \models \mathrm{clone}\ a : \alpha$.

Proof sketch. The proof is similar to that of Lemma A.6 (Method update). Assuming $\sigma :_{k,\Psi} \Sigma$ and $h :_k \Psi$ such that $\langle h, \mathrm{clone}\ \sigma(a)\rangle$ halts in fewer than k steps, by appealing to the operational semantics and the assumption that $\Sigma \models a : \alpha$ one obtains the existence of some Ψ^* such that

$$\langle h, \mathrm{clone}\ \sigma(a)\rangle \to^j \langle h^*, \mathrm{clone}\ \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \to \langle h', b'\rangle$$

with $(k, \Psi) \sqsubseteq (k-j, \Psi^*)$, $h^* :_{k-j} \Psi^*$ and $\langle k-j, \Psi^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha$. Since the only reduction from $\langle h^*, \mathrm{clone}\ \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle$ is by (Red-Clone) it is clear that for some (distinct) $l'_e \notin dom(h^*)$ we have

$$h' \;=\; h^*\left[l'_e := h^*(l_e)\right]_{e\in E} \quad\wedge\quad b' \;=\; \left\{\mathrm{m}_e{=}l'_e\right\}_{e\in E} \tag{A.35}$$

If we set $\Psi' = \left\lfloor \Psi^*\left[l'_e := \Psi^*(l_e)\right]_{e\in E} \right\rfloor_{k-j-1}$ then it follows that $(k, \Psi) \sqsubseteq (k-j-1, \Psi')$, and to establish the lemma it suffices to prove

$$h' :_{k-j-1} \Psi' \quad\wedge\quad \langle k-j-1, \Psi', \left\{\mathrm{m}_e{=}l'_e\right\}_{e\in E}\rangle \in \alpha \tag{A.36}$$

Observing that $dom(\Psi') \subseteq dom(h')$ is satisfied, the first conjunct is proved by showing that $\langle i, \lfloor \Psi' \rfloor_i, h'(l)\rangle \in \lfloor \Psi'(l) \rfloor_{k-j-1}$ holds for all $i < k-j-1$ and all $l \in dom(\Psi')$. This is done by a case distinction on whether $l \in dom(\Psi^*)$ or $l = l'_e$ for some e. In both cases, the relation follows from $h^* :_{k-j} \Psi^*$ and the closure under state extension of types.

As for the second conjunct of (A.36), we note that Ψ' is constructed from Ψ^* such that $\lfloor \Psi'(l'_e) \rfloor_{k-j-1} = \lfloor \Psi^*(l_e) \rfloor_{k-j-1}$ holds for all $e \in E$. Therefore, by unfolding the definition of the object types for $\langle k-j, \Psi^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha$, condition (Obj-3) allows

us to conclude that $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, \{\mathrm{m}_e{=}l'_e\}_{e\in E}\rangle \in \lfloor \alpha \rfloor_{k-j-1}$. Then the required $\langle k-j-1, \Psi', \{\mathrm{m}_e{=}l'_e\}_{e\in E}\rangle \in \alpha$ follows from the fact that $\lfloor \Psi' \rfloor_{k-j-1} = \Psi'$ holds by definition of Ψ', and that $\lfloor \alpha \rfloor_{k-j-1} \subseteq \alpha$. □

A.3. **Subtyping Lemmas for Object Types.**

Lemma A.8 (SemSubObj: Subtyping object types)**.** *$E \subseteq D$ and for all $e \in E$ if $\nu_e \in \{+, \circ\}$ then $\alpha_e \subseteq \beta_e$ and if $\nu_e \in \{-, \circ\}$ then $\beta_e \subseteq \alpha_e$ imply that $[m_d :_{\nu_d} \alpha_d]_{d\in D} \subseteq [m_e :_{\nu_e} \beta_e]_{e\in E}$.*

Proof. We denote $\alpha = [\mathrm{m}_d :_{\nu_d} \alpha_d]_{d\in D}$ and $\beta = [\mathrm{m}_e :_{\nu_e} \beta_e]_{e\in E}$. We assume that $E \subseteq D$ and

$$\forall e \in E.\ (\nu_e \in \{+, \circ\} \Rightarrow \alpha_e \subseteq \beta_e)\ \wedge\ (\nu_e \in \{-, \circ\} \Rightarrow \beta_e \subseteq \alpha_e), \tag{A.37}$$

and prove that for all heap typings Ψ, for all values v and all $k \geq 0$, if $\langle k, \Psi, v\rangle \in \alpha$ then $\langle k, \Psi, v\rangle \in \beta$, by complete induction on k. The induction hypothesis is that for all $j < k$ if $\langle j, \Psi, v\rangle \in \alpha$ then $\langle j, \Psi, v\rangle \in \beta$, or equivalently $\lfloor \alpha \rfloor_k \subseteq \lfloor \beta \rfloor_k$.

If we assume that $\langle k, \Psi, v\rangle \in \alpha$, then by the definition of α (Definition 3.20) we have that $v = \{\mathrm{m}_c{=}l_c\}_{c\in C}$, $D \subseteq C$ and there exists $\alpha' \in \mathit{Type}$ such that $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ and

$$\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \alpha_d) \tag{A.38}$$

Moreover, condition (Obj-3) holds with respect to α, *i.e.*, for all $j < k$, all Ψ' and all $\{\mathrm{m}_e{=}l'_e\}_{e\in E}$ such that $(k, \Psi) \sqsubseteq (j, \Psi')$,

$$(\forall e \in E.\ \lfloor \Psi' \rfloor_j (l'_e) = \lfloor \Psi \rfloor_j (l_e)) \ \Rightarrow\ \langle j, \lfloor \Psi' \rfloor_j, \{\mathrm{m}_e{=}l'_e\}_{e\in E}\rangle \in \alpha' \tag{A.39}$$

From $E \subseteq D$ and $D \subseteq C$ by transitivity $E \subseteq C$. From $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ and the induction hypothesis $\lfloor \alpha \rfloor_k \subseteq \lfloor \beta \rfloor_k$ we get that $\lfloor \alpha' \rfloor_k \subseteq \lfloor \beta \rfloor_k$, *i.e.*, (Obj-1) holds. Moreover, (A.39) entails that condition (Obj-3) also holds with respect to the object type β. So in order to conclude that $\langle k, \Psi, v\rangle \in \beta$, and therefore that $\alpha \subseteq \beta$, all that remains to be proven is condition (Obj-2):

$$\forall e \in E.\ \langle k, \Psi, l_e\rangle \in \mathrm{ref}_{\nu_e}(\alpha' \to \beta_e)$$

For this, we choose some e in E and do a case analysis on the variance annotation ν_e:

- Case $\nu_e = +$. By (A.37) we deduce that $\alpha_e \subseteq \beta_e$, thus by the covariance of the procedure type constructor in its second argument (SemSubProc in Figure 6) we get that $\alpha' \to \alpha_e \subseteq \alpha' \to \beta_e$. But since $E \subseteq D$ from (A.38) we know that $\langle k, \Psi, l_e\rangle \in \mathrm{ref}_+(\alpha' \to \alpha_e)$. Since the type constructor ref_+ is covariant (SemSubCovRef in Figure 7) this implies $\langle k, \Psi, l_e\rangle \in \mathrm{ref}_+(\alpha' \to \beta_e)$.
- Case $\nu_e = -$. Similarly to the previous case, (A.37) gives us that $\beta_e \subseteq \alpha_e$. Again by the covariance of $\lambda\xi.\, \alpha' \to \xi$ (SemSubProc) we infer that $\alpha' \to \beta_e \subseteq \alpha' \to \alpha_e$. From (A.38) $\langle k, \Psi, l_e\rangle \in \mathrm{ref}_-(\alpha' \to \alpha_e)$, so by the contravariance of ref_- (SemSubConRef in Figure 7) we get that $\langle k, \Psi, l_e\rangle \in \mathrm{ref}_-(\alpha' \to \beta_e)$.
- $\nu_e = \circ$. Now (A.37) entails that $\alpha_e = \beta_e$. Since $\langle k, \Psi, l_e\rangle \in \mathrm{ref}_\circ(\alpha' \to \alpha_e)$ by (A.38) we immediately obtain that also $\langle k, \Psi, l_e\rangle \in \mathrm{ref}_\circ(\alpha' \to \beta_e)$. □

Lemma A.9 (SemSubObjVar: Subtyping object variances)**.** *If for all $d \in D$ we have $\nu_d = \circ$ or $\nu_d = \nu'_d$ then $[m_d :_{\nu_d} \tau_d]_{d\in D} \subseteq [m_d :_{\nu'_d} \tau_d]_{d\in D}$.*

Proof. The proof proceeds similarly to the proof of Lemma A.8. Let us denote $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ and $\alpha' = [\mathrm{m}_d :_{\nu'_d} \tau_d]_{d\in D}$. We assume that

$$\forall d \in D.\ \nu_d = \circ \ \vee\ \nu_d = \nu'_d \tag{A.40}$$

Let Ψ and v be arbitrary. We prove that for all $k \geq 0$, if $\langle k, \Psi, v\rangle \in \alpha$ then $\langle k, \Psi, v\rangle \in \alpha'$, by complete induction on k. The induction hypothesis is that for all $j < k$ if $\langle j, \Psi, v\rangle \in \alpha$ then $\langle j, \Psi, v\rangle \in \alpha'$, or equivalently $\lfloor\alpha\rfloor_k \subseteq \lfloor\alpha'\rfloor_k$.

Assume that $\langle k, \Psi, v\rangle \in \alpha$, then by the definition of α we have that $v = \{\mathrm{m}_e{=}l_e\}_{e\in E}$, $D \subseteq E$, and there exists a type α'' such that $\lfloor\alpha''\rfloor_k \subseteq \lfloor\alpha\rfloor_k$ and

$$\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha'' \to \tau_d) \tag{A.41}$$

Moreover, condition (Obj-3) holds.

From $\lfloor\alpha''\rfloor_k \subseteq \lfloor\alpha\rfloor_k$ and the induction hypothesis $\lfloor\alpha\rfloor_k \subseteq \lfloor\alpha'\rfloor_k$ by transitivity we get that $\lfloor\alpha''\rfloor_k \subseteq \lfloor\alpha'\rfloor_k$. This choice of α'' also shows that (Obj-3) holds for $\langle k, \Psi, v\rangle$ with respect to α'. So in order to show that $\langle k, \Psi, v\rangle \in \alpha'$, and therefore that $\alpha \subseteq \alpha'$, all that remains to be proven is that:

$$\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu'_d}(\alpha'' \to \tau_d)$$

We show this by case analysis on the disjunction in (A.40). Both cases are trivial:

- Case $\nu_d = \circ$. From (A.41) and $\mathrm{ref}_\circ(\alpha'' \to \tau_d) \subseteq \mathrm{ref}_{\nu'_d}(\alpha'' \to \tau_d)$ (SemSubVarRef in Figure 7) it is immediate that $\langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu'_d}(\alpha'' \to \tau_d)$.
- Case $\nu_d = \nu'_d$, then the required statement is the same as (A.41). □

A.4. **Typing Lemmas with Structural Assumptions for Self Types.**

Lemma A.10 (SemUpd-Str: Method update with structural assumptions)**.** *For all object types $\alpha = [m_d :_{\nu_d} F_d]_{d\in D}$ and all $\alpha' \in$ Type such that $\alpha' \lhd \alpha$, if $e \in D$, $\nu_e \in \{-, \circ\}$ and $\Sigma \models a : \alpha'$ and $\Sigma[x := \alpha'] \models b : F_e(\alpha')$, then $\Sigma \models a.m_e := \varsigma(x)b : \alpha'$.*

Proof. The proof is an adaptation of the proof given for Lemma A.6 (Method update) above. Assume $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D}$, $e \in D$, and $\nu_e \in \{-, \circ\}$, and let $\alpha' \in$ *Type* such that $\alpha' \lhd \alpha$. Moreover assume that $\Sigma \models a : \alpha'$ and $\Sigma[x := \alpha'] \models b : F_e(\alpha')$ hold. We show that $\Sigma \models a.\mathrm{m}_e := \varsigma(x)b : \alpha'$.

Let $k \geq 0$, σ be a value environment and Ψ be a heap typing such that $\sigma :_{k,\Psi} \Sigma$. We must prove that $\sigma(a.\mathrm{m}_e := \varsigma(x)b) :_{k,\Psi} \alpha'$, so let h and $j < k$ be such that

$$h :_k \Psi \quad\wedge\quad \langle h, \sigma(a).\mathrm{m}_e := \varsigma(x)\sigma(b)\rangle \to^j \langle h', a'\rangle \quad\wedge\quad \langle h', a'\rangle \not\to \tag{A.42}$$

By the operational semantics, this sequence is induced by $\langle h, \sigma(a)\rangle \to^i \langle h'', a''\rangle$ for $i \leq j$ and some h'' and a'', and by the assumption $\Sigma \models a : \alpha'$ there exists some Ψ'' such that

$$(k, \Psi) \sqsubseteq (k - i, \Psi'') \quad\wedge\quad h'' :_{k-i} \Psi'' \quad\wedge\quad \langle k - i, \Psi'', a''\rangle \in \alpha' \subseteq [\mathrm{m}_d :_{\nu_d} F_d]_{d\in D} \tag{A.43}$$

In particular, a'' is of the form $\{\mathrm{m}_e{=}l_e\}_{e\in E}$ for some $E \supseteq D$, and by the operational semantics $\langle h'', a''.\mathrm{m}_e := \varsigma(x)\sigma(b)\rangle \to \langle h', a''\rangle$. In particular, a' is a'' and h' is $h''[l_e := \lambda(x)\sigma(b)]$. By choosing $\Psi' = \lfloor\Psi''\rfloor_{k-j}$, the first and last conjuncts of (A.43) yield

$$(k, \Psi) \sqsubseteq (k - j, \Psi') \quad\wedge\quad \langle k - j, \Psi', a''\rangle \in \alpha'$$

by Proposition A.2 and transitivity, and by closure under state extension of α'. To establish the lemma, it remains to show that $h' :_k \Psi'$. For $l \in dom(\Psi') - \{l_e\}$ this follows from the second conjunct of (A.43) by the closure under state extension. The interesting case is when $l = l_e$ and we must prove $h'(l) = \lambda(x)\sigma(b) :_{k-j} \Psi'(l)$. Since $\alpha' \lhd \alpha$ and $\langle k-j, \Psi', a'' \rangle \in \alpha'$, condition (Obj-2-self) in Definition 5.2 (Self type exposure) yields $\langle k-j, \Psi', l_e \rangle \in \mathrm{ref}_{\nu_e}(\alpha' \to F_e(\alpha'))$. By assumption, $\nu_e \in \{-, \circ\}$ so $\Psi'(l) \supseteq \lfloor \alpha' \to F_e(\alpha') \rfloor_{k-j}$ holds by the definition of ref_{ν_e}. Hence it suffices to prove that

$$\lambda(x)\sigma(b) :_{k-j,\Psi'} \alpha' \to F_e(\alpha')$$

which follows from the assumption $\Sigma[x := \alpha'] \models b : F_e(\alpha')$. □

Proposition A.11 (Self type exposure)**.** *Let α be a self type and suppose $\langle k, \Psi, v \rangle \in \alpha$. Then there exists $\alpha'' \in$ Type such that $\alpha'' \lhd \alpha$ and $\langle k-1, \lfloor \Psi \rfloor_{k-1}, v \rangle \in \alpha''$.*

Proof. Suppose $\langle k, \Psi, v \rangle \in \alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$. By Definition 5.1 (Self types), this means that there exists $\alpha' \in$ *Type* such that $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ and conditions (Obj-2-Self) and (Obj-3) are satisfied. Choosing $\alpha'' = \lfloor \alpha' \rfloor_k$, it is clear that $\alpha'' \lhd \alpha$ since all the conditions only rely on α' to approximation k. Moreover, by instantiating $\Psi' = \Psi$ and $\{\mathrm{m}_e = l'_e\}_{e \in E} = v$ in (Obj-3) we obtain that $\langle k-1, \lfloor \Psi \rfloor_{k-1}, v \rangle \in \alpha''$. This proves the proposition. □

Lemma A.12 (SemLet-Str: Introducing structural assumptions)**.** *Let $\alpha = [m_d :_{\nu_d} F_d]_{d \in D}$ and suppose that $\Sigma \models a : \alpha$ and that $\Sigma[x := \xi] \models b : \beta$ for all $\xi \in$ Type with $\xi \lhd \alpha$. Then $\Sigma \models \mathrm{let}\ x = a\ \mathrm{in}\ b : \beta$.*

Proof. Let $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$ and suppose that $\Sigma \models a : \alpha$ and that $\Sigma[x := \xi] \models b : \beta$ for all $\xi \in$ *Type* with $\xi \lhd \alpha$. We must show that $\Sigma \models \mathrm{let}\ x = a\ \mathrm{in}\ b : \beta$. Thus, let $k \geq 0$, Ψ and σ be such that $\sigma :_{k,\Psi} \Sigma$. By the definition of the semantic typing judgement (Definition 3.8) we must show that $\sigma(\mathrm{let}\ x = a\ \mathrm{in}\ b) :_{k,\Psi} \beta$, or equivalently (after suitable α-renaming and removing the syntactic sugar) that

$$(\lambda(x)\sigma(b))\ \sigma(a) :_{k,\Psi} \beta$$

Suppose $j < k, h, h'$ and b' are such that

$$h :_k \Psi \quad \wedge \quad \langle h, (\lambda(x)\sigma(b))\ \sigma(a) \rangle \to^j \langle h', b' \rangle \quad \wedge \quad \langle h', b' \rangle \not\to \tag{A.44}$$

From the second and third conjunct of (A.44) by the operational semantics we have that for some $i \leq j < k$, some h'' and some Ψ'',

$$\langle h, \sigma(a) \rangle \to^i \langle h'', a'' \rangle \not\to \quad \wedge \quad \langle h'', (\lambda(x)\sigma(b))\ a'' \rangle \to^{j-i} \langle h', b' \rangle \tag{A.45}$$

From the first conjunct together with the assumption $\Sigma \models a : \alpha$ and the first conjunct of (A.44), by Definition 3.6 it follows that there exists a heap typing Ψ'' such that

$$(k, \Psi) \sqsubseteq (k-i, \Psi'') \quad \wedge \quad h'' :_{k-i} \Psi'' \quad \wedge \quad \langle k-i, \Psi'', a'' \rangle \in \alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D} \tag{A.46}$$

In particular, $a'' \in$ *Val* and the operational semantics gives

$$\langle h, (\lambda(x)\sigma(b))\ (\sigma(a)) \rangle \to^i \langle h'', (\lambda(x)\sigma(b))\ a'' \rangle \to \langle h'', \sigma\big[x := a''\big](b) \rangle \to^{j-i-1} \langle h', b' \rangle \tag{A.47}$$

From the third conjunct of (A.46) by Proposition A.11 there exists $\alpha' \in$ *Type* such that $\alpha' \lhd \alpha$ and $\langle k-i-1, \lfloor \Psi'' \rfloor_{k-i-1}, a'' \rangle \in \alpha'$. From $\sigma :_{k,\Psi} \Sigma$, the first conjunct of (A.46), Propositions A.1 and A.2 and the closure under state extension, this yields

$$\sigma\big[x := a''\big] :_{k-i-1, \lfloor \Psi'' \rfloor_{k-i-1}} \Sigma\big[x := \alpha'\big] \tag{A.48}$$

Since $\alpha' \lhd \alpha$, by instantiating the universally quantified type ξ in the hypothesis on b we obtain that $\Sigma[x := \alpha'] \models b : \beta$. Therefore, (A.48) gives $\sigma[x := a''](b) :_{k-i-1, \lfloor \Psi'' \rfloor_{k-i-1}} \beta$. Clearly $h'' :_{k-i-1} \lfloor \Psi'' \rfloor_{k-i-1}$ by the second conjunct of (A.46), so that the second conjunct of (A.45) shows that there is some Ψ' such that $(k, \Psi) \sqsubseteq (k-i-1, \lfloor \Psi'' \rfloor_{k-i-1}) \sqsubseteq (k-j, \Psi')$, $h' :_{k-j} \Psi'$ and $\langle k-j, \Psi', b' \rangle \in \beta$, by Definition 3.6. This establishes that $\sigma(\text{let } x = a \text{ in } b) :_{k,\Psi} \beta$ holds as required. □

We next define a recursive type of records $\{\!|\, \mathrm{m}_d :_{\nu_d} F_d \,|\!\}_{d \in D}$, which is the type arising from the recursive record interpretation of (imperative) objects [18]. While this type does not give rise to non-trivial subtyping, we will show that it satisfies $\{\!|\, \mathrm{m}_d :_{\nu_d} F_d \,|\!\}_{d \in D} \lhd [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$.

Definition A.13. Assume $F_d : \mathit{Type} \to \mathit{Type}$ are monotonic and non-expansive type constructors, for all $d \in D$. Then let $\beta = \{\!|\, \mathrm{m}_d :_{\nu_d} F_d \,|\!\}_{d \in D}$ be defined as the set of all triples $\langle k, \Psi, \{\mathrm{m}_d = l_d\}_{d \in D} \rangle$ such that

$$(\forall d \in D.\ \langle k, \Psi, l_d \rangle \in \mathrm{ref}_{\nu_d}(\beta \to F_d(\beta))) \tag{Rec-1}$$

$$\wedge \quad (\forall j < k.\ \forall \Psi'.\ \forall \left\{\mathrm{m}_e = l'_e\right\}_{e \in E}. \tag{Rec-2}$$

$$(k, \Psi) \sqsubseteq (j, \Psi') \ \wedge\ (\forall d \in D.\ \left\lfloor \Psi' \right\rfloor_j (l'_d) = \lfloor \Psi \rfloor_j (l_d)) \Rightarrow \langle j, \left\lfloor \Psi' \right\rfloor_j, \left\{\mathrm{m}_d = l'_d\right\}_{d \in D} \rangle \in \beta)$$

Note that the recursive specification of β is well-founded, *i.e.*, β is well-defined. Moreover, β is a type, *i.e.*, it is closed under state extension.

Proposition A.14. *For all self types $[m_d :_{\nu_d} F_d]_{d \in D}$ we have that*

$$\{\!|\, m_d :_{\nu_d} F_d \,|\!\}_{d \in D} \lhd [m_d :_{\nu_d} F_d]_{d \in D}$$

Proof. Let $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$ and $\beta = \{\!|\, \mathrm{m}_d :_{\nu_d} F_d \,|\!\}_{d \in D}$ for some arbitrary monotonic and non-expansive type constructors F_d. It is clear that for all $\langle k, \Psi, \{\mathrm{m}_d = l_d\}_{d \in D} \rangle \in \beta$, conditions (Obj-2-Self) and (Obj-3) from Definition 5.2 are satisfied, by the definition of β (Definition A.13). It remains to prove that $\beta \subseteq \alpha$. We establish this by showing that for all $k \geq 0$, $\lfloor \beta \rfloor_k \subseteq \lfloor \alpha \rfloor_k$, by complete induction on k. Let $\langle k, \Psi, \{\mathrm{m}_d = l_d\}_{d \in D} \rangle \in \beta$; we need to show that $\langle k, \Psi, \{\mathrm{m}_d = l_d\}_{d \in D} \rangle \in \alpha$. We have that $D \subseteq D$ and we choose $\alpha' = \beta$ which is a type and fulfills $\lfloor \beta \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ (Obj-1) by the induction hypothesis. The conditions (Obj-2-Self) and (Obj-3) in Definition 5.1 (Self types) are exactly the same as conditions (Rec-1) and (Rec-2) in the definition of β (Definition A.13), which concludes the proof. □

Lemma A.15 (SemObj-Str: Object construction with structural assumptions)**.** *Let $\alpha = [m_d :_{\nu_d} F_d]_{d \in D}$ and suppose that for all $d \in D$ and all $\xi \in \mathit{Type}$ with $\xi \lhd \alpha$, $\Sigma[x := \xi] \models b_d : F_d(\xi)$. Then $\Sigma \models [m_d = \varsigma(x_d) b_d]_{d \in D} : \alpha$.*

Proof. Let $\alpha = [\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$ and assume that

$$\forall d \in D.\ \forall \xi \in \mathit{Type}.\ \xi \lhd \alpha \Rightarrow \Sigma[x_d := \xi] \models b_d : F_d(\xi) \tag{A.49}$$

We must show that $\Sigma \models [\mathrm{m}_d = \varsigma(x_d) b_d]_{d \in D} : \alpha$. Thus, let $k \geq 0$, σ be a value environment and Ψ be a heap typing such that $\sigma :_{k,\Psi} \Sigma$. By Definition 3.8 we need to show that $\sigma([\mathrm{m}_d = \varsigma(x_d) b_d]_{d \in D}) :_{k,\Psi} \alpha$. Equivalently (after suitable α-renaming), we show that

$$[\mathrm{m}_d = \varsigma(x_d) \sigma(b_d)]_{d \in D} :_{k,\Psi} \alpha$$

Suppose $j < k, h, h'$ and b' are such that the following three conditions are fulfilled:

$$h :_k \Psi \quad \wedge \quad \langle h, [\mathrm{m}_d = \varsigma(x_d) \sigma(b_d)]_{d \in D} \rangle \to^j \langle h', b' \rangle \quad \wedge \quad \langle h', b' \rangle \not\to \tag{A.50}$$

By the operational semantics RED-OBJ is the only rule that applies, which means that necessarily $j = 1$ and for some distinct $l_d \notin dom(h)$ we have $b' = \{\mathrm{m}_d{=}l_d\}_{d\in D}$ and

$$h' = h\,[l_d := \lambda(x_d)\sigma(b_d)]_{d\in D} \tag{A.51}$$

Let $\beta = \{\!|\, \mathrm{m}_d :_{\nu_d} F_d \,|\!\}_{d\in D}$, as in Definition A.13. We choose

$$\Psi' = \lfloor \Psi\,[l_d := (\beta \to F_d(\beta))]_{d\in D} \rfloor_{k-1} \tag{A.52}$$

and show that

$$(k,\Psi) \sqsubseteq (k-1,\Psi') \quad\wedge\quad h' :_{k-1} \Psi' \quad\wedge\quad \langle k-1, \Psi', b'\rangle \in \alpha \tag{A.53}$$

The first conjunct of (A.53) holds by the construction of Ψ' (A.52). In order to show the second conjunct, let $i < k-1$ and $l \in dom(\Psi')$. We now need to show that $\langle i, \lfloor\Psi'\rfloor_i, h'(l)\rangle \in \Psi'(l)$. In case $l \in dom(\Psi)$ the proof proceeds exactly as for Lemma A.4, so we only consider the case when $l = l_d$ for some $d \in D$. From (A.51) and (A.52) we get that

$$h'(l) = \lambda(x_d)\sigma(b_d) \quad\wedge\quad \Psi'(l) = \lfloor \beta \to F_d(\beta)\rfloor_{k-1}$$

Thus we need to show that

$$\langle i, \lfloor\Psi'\rfloor_i, \lambda(x_d)\sigma(b_d)\rangle \in \lfloor \beta \to F_d(\beta)\rfloor_{k-1} \tag{A.54}$$

By Proposition A.14 we obtain that $\beta \lhd \alpha$, so we can instantiate the universally quantified ξ in (A.49) with β and obtain that

$$\Sigma[x_d := \beta] \models b_d : F_d(\beta)$$

By SEMLAM in Figure 6 (Lemma 3.15) this gives us that

$$\Sigma \models \lambda(x_d) b_d : \beta \to F_d(\beta)$$

From this and $\sigma :_{k,\Psi} \Sigma$ by Definition 3.8 we obtain

$$\lambda(x_d)\sigma(b_d) \;:_{k,\Psi}\; \beta \to F_d(\beta) \tag{A.55}$$

Since $k > 1$ and from (A.50) $h :_k \Psi$, Proposition A.3 shows that (A.55) implies

$$\langle k, \Psi, \lambda(x_d)\sigma(b_d)\rangle \in \beta \to F_d(\beta) \tag{A.56}$$

By Proposition A.2 we get that $(k-1,\Psi') \sqsubseteq (i, \lfloor\Psi'\rfloor_i)$, which together with the first conjunct of (A.53) and the transitivity of $\sqsubseteq$ yields $(k,\Psi) \sqsubseteq (i, \lfloor\Psi'\rfloor_i)$. Since each $\beta \to F_d(\beta)$ is closed under state extension, the latter property and (A.56) imply the required (A.54).

Finally, we need to show the third conjunct of (A.53), *i.e.*, $\langle k-1, \Psi', \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle \in \alpha$. To this end, we prove the following more general claim:
Claim: For all $j_0 \geq 0$, for all Ψ_0 and for all $\{\mathrm{m}_d{=}l'_d\}_{d\in D}$

$$\begin{aligned}(k-1,\Psi') \sqsubseteq (j_0,\Psi_0) \;\wedge\; (\forall d \in D.\; \lfloor\Psi_0\rfloor_{j_0}(l'_d) = \lfloor\Psi'\rfloor_{j_0}(l_d)) \\ \Rightarrow\; \langle j_0, \lfloor\Psi_0\rfloor_{j_0}, \{\mathrm{m}_d{=}l'_d\}_{d\in D}\rangle \in \beta\end{aligned} \tag{A.57}$$

From this and $\lfloor\Psi'\rfloor_{k-1} = \Psi'$, the last conjunct of (A.53) follows by taking $j_0 = k-1$, $\Psi_0 = \Psi'$, and $l'_d = l_d$ for all $d \in D$, and by observing that $\sqsubseteq$ is reflexive (Proposition A.1) and $\beta \subseteq \alpha$ (since $\beta \lhd \alpha$).

The claim above is proved by complete induction on j_0. So assume $j_0 \geq 0$ and Ψ_0 are such that

$$(k-1,\Psi') \sqsubseteq (j_0,\Psi_0) \tag{A.58}$$

Moreover, for all $d \in D$ let $l'_d \in dom(\Psi_0)$ such that

$$\lfloor \Psi_0 \rfloor_{j_0}(l'_d) = \lfloor \Psi' \rfloor_{j_0}(l_d) \tag{A.59}$$

We show that $\langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, \{\mathrm{m}_d{=}l'_d\}_{d\in D}\rangle \in \beta$, by checking that the two conditions from the definition of β (Definition A.13) are satisfied. By the construction of Ψ' in (A.52), together with (A.58) and (A.59), it follows that for all $d \in D$

$$\lfloor \Psi_0 \rfloor_{j_0}(l'_d) = \lfloor \Psi' \rfloor_{j_0}(l_d) = \lfloor \beta \to F_d(\beta) \rfloor_{j_0} \tag{A.60}$$

By the definition of reference types (Definition 3.16) this implies that

$$\forall d \in D.\ \langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, l'_d\rangle \in \mathrm{ref}_\circ(\beta \to F_d(\beta)) \tag{A.61}$$

By the lemma for subtyping reference types (SemSubVarRef in Figure 7) we then obtain property (Rec-1):

$$\forall d \in D.\ \langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, l'_d\rangle \in \mathrm{ref}_{\nu_d}(\beta \to F_d(\beta)) \tag{A.62}$$

Second, we must prove (Rec-2), *i.e.*, that for all $j < j_0$, Ψ_1 and $\{\mathrm{m}_d{=}l''_d\}_{d\in D}$

$$(j_0, \Psi_0) \sqsubseteq (j, \Psi_1) \ \wedge\ (\forall d \in D.\ \lfloor \Psi_1 \rfloor_j (l''_d) = \lfloor \Psi_0 \rfloor_j (l'_d)) \\ \Rightarrow\ \langle j, \lfloor \Psi_1 \rfloor_j, \{\mathrm{m}_d{=}l''_d\}_{d\in D}\rangle \in \beta \tag{A.63}$$

Note that this last condition holds vacuously in the base case of the induction, when $j_0 = 0$. So assume $j < j_0$ and Ψ_1 and l''_d are such that $(j_0, \Psi_0) \sqsubseteq (j, \Psi_1)$ and $\lfloor \Psi_1 \rfloor_j (l''_d) = \lfloor \Psi_0 \rfloor_j (l'_d)$ for all $d \in D$. Now $j < j_0$ and assumption (A.59) yield that for all $d \in D$

$$\lfloor \Psi_1 \rfloor_j (l''_d) = \lfloor \Psi_0 \rfloor_j (l'_d) = \Big\lfloor \lfloor \Psi_0 \rfloor_{j_0} (l'_d) \Big\rfloor_j = \Big\lfloor \lfloor \Psi' \rfloor_{j_0} (l_d) \Big\rfloor_j = \lfloor \Psi' \rfloor_j (l_d)$$

Moreover, from $(k-1, \Psi') \sqsubseteq (j_0, \Psi_0)$ (A.58) and $(j_0, \Psi_0) \sqsubseteq (j, \Psi_1)$, by the transitivity of $\sqsubseteq$ we have that $(k-1, \Psi') \sqsubseteq (j, \Psi_1)$. Since $j < j_0$, the induction hypothesis of the claim gives

$$\langle j, \lfloor \Psi_1 \rfloor_j, \{\mathrm{m}_d{=}l''_d\}_{d\in D}\rangle \in \beta$$

and we have established (A.63).

By Definition A.13 applied to the type $\beta = \{\!|\, \mathrm{m}_d :_{\nu_d} F_d \,|\!\}_{d\in D}$ the properties (A.62), and (A.63) establish that indeed $\langle j_0, \lfloor \Psi_0 \rfloor_{j_0}, \{\mathrm{m}_d{=}l'_d\}_{d\in D}\rangle \in \beta$. This finishes the inductive proof of claim (A.57), and the proof of the lemma. □

A.5. **Subtyping Lemma for Generalized Object Types.**

Lemma A.16 (SemSubGen-Obj: Subtyping generalized object types)**.** *If $E \subseteq D$ and for all $e \in E$ we have that $\beta^w_e \subseteq \alpha^w_e$ and $\alpha^r_e \subseteq \beta^r_e$ then $[m_d : (\alpha^w_d, \alpha^r_d)\,]_{d\in D} \subseteq [m_e : (\beta^w_e, \beta^r_e)\,]_{e\in E}$.*

Proof. Denote $\alpha = [\mathrm{m}_d : (\alpha^w_d, \alpha^r_d)\,]_{d\in D}$, $\beta = [\mathrm{m}_e : (\beta^w_e, \beta^r_e)\,]_{e\in E}$, and assume $E \subseteq D$ and

$$\forall e \in E.\ (\beta^w_e \subseteq \alpha^w_e \ \wedge\ \alpha^r_e \subseteq \beta^r_e). \tag{A.64}$$

We prove that for all heap typings Ψ, for all values v and all $k \geq 0$, if $\langle k, \Psi, v\rangle \in \alpha$ then $\langle k, \Psi, v\rangle \in \beta$, by complete induction on k. The induction hypothesis is that $\lfloor \alpha \rfloor_k \subseteq \lfloor \beta \rfloor_k$.

If we assume that $\langle k, \Psi, v\rangle \in \alpha$, then by the definition of α (Definition 3.20 with condition (Obj-2-Gen) instead of (Obj-2)) we have that $v = \{\mathrm{m}_c{=}l_c\}_{c\in C}$, $D \subseteq C$ and there exists $\alpha' \in \mathit{Type}$ such that $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ and

$$\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}(\alpha' \to \alpha^w_d, \alpha' \to \alpha^r_d) \tag{A.65}$$

Moreover, condition (Obj-3) holds with respect to α, *i.e.*, for all $j < k$, all Ψ' and all $\{\mathrm{m}_e{=}l'_e\}_{e\in E}$ such that $(k,\Psi) \sqsubseteq (j,\Psi')$,

$$(\forall e \in E.\ \lfloor \Psi' \rfloor_j (l'_e) = \lfloor \Psi \rfloor_j (l_e)) \ \Rightarrow\ \langle j, \lfloor \Psi' \rfloor_j, \{\mathrm{m}_e{=}l'_e\}_{e\in E} \rangle \in \alpha' \tag{A.66}$$

From $E \subseteq D$ and $D \subseteq C$ by transitivity $E \subseteq C$. From $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ and the induction hypothesis $\lfloor \alpha \rfloor_k \subseteq \lfloor \beta \rfloor_k$ we get that $\lfloor \alpha' \rfloor_k \subseteq \lfloor \beta \rfloor_k$, *i.e.*, (Obj-1) holds. Moreover, (A.66) entails that condition (Obj-3) also holds with respect to the object type β. So in order to conclude that $\langle k, \Psi, v \rangle \in \beta$, all that remains to be proven is condition (Obj-2-Gen):

$$\forall e \in E.\ \langle k, \Psi, l_e \rangle \in \mathrm{ref}(\alpha' \to \beta^w_e, \alpha' \to \beta^r_e) \tag{A.67}$$

Let $e \in E$. Since the procedure type constructor is covariant in the result type (SemSubProc in Figure 6) assumption A.64 implies that

$$\alpha' \to \beta^w_e \subseteq \alpha' \to \alpha^w_e \ \wedge\ \alpha' \to \alpha^r_e \subseteq \alpha' \to \beta^r_e$$

From this by (SemSubRef-Gen) we get that

$$\mathrm{ref}(\alpha' \to \alpha^w_e, \alpha' \to \alpha^r_e) \subseteq \mathrm{ref}(\alpha' \to \beta^w_e, \alpha' \to \beta^r_e)$$

This together with A.65 and $E \subseteq D$ directly implies A.67, which concludes the proof. □